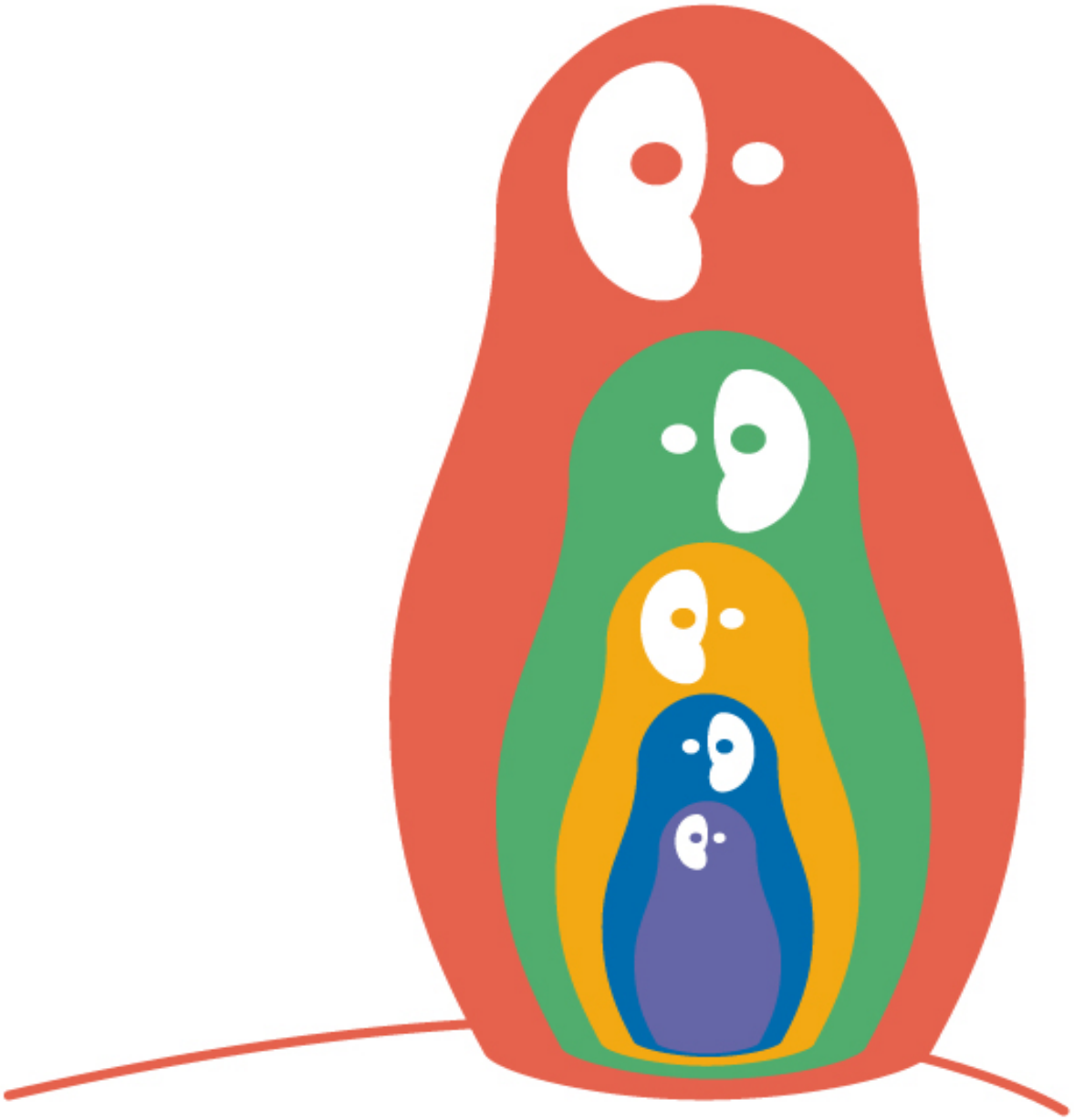
JavaScript Assertivo

Testes e qualidade de código em todas as camadas da aplicação



Casa do Código | alura

GABRIEL RAMOS

Sumário

- ISBN
- Agradecimentos
- Prefácio por Willian Justen
- Sobre o autor
- Todo mundo na mesma página
- Parte 1: Fundamentos de testes
  - 1 Uma conversa (nem tão) séria sobre testes
  - 2 Análise estática de código com ESLint
  - 3 Simulando um framework de testes
  - 4 Diga olá ao Jest!
- Parte 2: Aplicando testes unitários em uma CLI
  - 5 Testando código síncrono
  - 6 Testando código assíncrono
  - 7 Ajustando configurações e testando middlewares
- Parte 3: Testando aplicações back-end
  - 8 Testes unitários com Node e Express
  - 9 Testes de integração na API de usuários
  - 10 Testes de carga
- Parte 4: Testando aplicações front-end
  - 11 Testes unitários nos componentes da aplicação
  - 12 Testes de integração nas telas da aplicação
  - 13 Testes de regressão visual
- Parte 5: Testando de ponta a ponta
  - 14 Testes de ponta a ponta (end-to-end)
- Parte 6: Extras e conteúdos relevantes após a nossa jornada
  - 15 Próximos passos nessa jornada
  - 16 Glossário

ISBN

Impresso: 978-65-86110-84-5

Digital: 978-65-86110-85-2

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

É impossível falar sobre agradecimentos sem pensar nas pessoas que apoiam sua jornada e em todos os degraus que você percorre.

Não posso fazer dedicações sem começar pelo meu irmão Sérgio Luiz, por ser sempre a maior referência que eu poderia ter, um espelho de ser humano, de profissional e de conhecimento.

À minha companheira Anna, por estar sempre ao meu lado desde o início, tanto figurativamente quanto literalmente.

Aos meus pais Lilian Cristina e Sérgio Luiz, tia Leila, tio Luciano e avó Elizabeth, por toda a base e qualquer construção que me transformaram na pessoa que eu sou hoje.

Aos meus amigos de tecnologia com os quais eu tenho a honra de aprender diariamente. Evito citar nomes para não esquecer ninguém, mas vocês sabem da importância e do impacto que possuem no código que eu escrevo.

Às alunas e aos alunos que tive e tenho o imenso prazer de conhecer nos projetos de que faço parte, pessoas com as quais eu, com certeza, aprendo muito mais do que ensino.

Não existiria um caractere neste livro se não fosse por essas pessoas. Nas páginas a seguir, tem um pedacinho de cada um de vocês.

> “I find your lack of tests disturbing.” - **VADER**, 1977. Lord Sith

Prefácio por Willian Justen

Toda operação é sujeita a falhas, desde a criação de uma aplicação web até uma fábrica de criação de peças para carros. E essas falhas podem custar desde algumas dezenas de milhares de reais até milhões! Na programação, nós demos o apelido carinhoso de "*bug*" para essas falhas. E assim como a área de desenvolvimento é bem diversa, esses erros também podem ser, e serão, bem diferentes, seja um botão que não executa nenhuma ação ao ser clicado ou um relatório gerado com valores errados, causando uma completa confusão para os seus clientes.

Alguns dados interessantes para entendermos as dimensões dos problemas que bugs podem causar:

- Em outubro de 2018 e março de 2019 aconteceram dois acidentes fatais envolvendo o modelo de avião Boeing 737 Max. Depois de uma extensa análise, foi descoberto que ambos os acidentes foram causados por uma falha em um software chamado *Mcas*, que impediu que os pilotos pudessem alterar o ângulo de inclinação da aeronave.
- Entre 2018 e 2019, a Nissan precisou fazer um recall de mais de 1 milhão de carros, pois o software da câmera de ré não resetava as configurações toda vez que o usuário iniciava a ré.
- Em outubro de 1999, uma nave espacial da NASA estimada em 125 milhões de dólares foi perdida no espaço devido a um erro de conversão de dados! O software utilizou os dados com o sistema de medidas americano em vez do sistema métrico, deixando a espaçonave fora de órbita.

Você talvez esteja pensando: "Mas por que isso me importa? Eu não trabalho na NASA!". Mas todos esses dados foram apresentados apenas para mostrar como falhas em software podem causar enormes danos. E não, essas coisas não acontecem somente em empresas enormes, mas até no e-commerce do *Seu Zé* da papelaria da esquina.

Pensando nessas falhas, nós precisamos estar preparados para eliminá-las antes mesmo de chegarem ao usuário final. Sendo assim, a melhor forma é testar sua aplicação e testar bastante. O problema é que testar todas as possibilidades nos toma tempo e, quanto maior a aplicação vai ficando, mais coisas podem ficar para trás e, assim, mais falhas podem ir aparecendo sem que nós possamos perceber. Outro grande problema do teste manual é que nós humanos também somos muito suscetíveis a falhas, ou seja, além das possíveis falhas de software, teremos que somar as falhas humanas.

Diante dessas dificuldades e problemas é que nasceram os testes automatizados de software, que são códigos testando outros códigos. Parece ser muito complicada essa metalinguagem e é um pouco difícil de se entender os conceitos mesmo, mas não se sinta mal se algumas coisas não se encaixarem automaticamente na sua cabeça. O que posso dizer seguramente é que este livro ajudará muito nos seus primeiros passos com os testes e, mesmo que você já tenha escrito testes antes, este livro vai ajudar você a escrevê-los melhor e a ter uma visão mais ampla em diferentes situações.

Eu estou na área há muitos anos e tenho orgulho de dizer que fui um dos primeiros a falar mais sobre testes de software no Brasil, principalmente na área de front-end, tendo já escrito sobre o assunto no meu blog, palestrado sobre o tema em diversos eventos e criado um dos poucos cursos de JavaScript focado em testes desde o início. Recebi com muita alegria o convite do Gabriel para escrever este prefácio, pois, além de ser sobre um tema que amo, já sabia de antemão que seria um livro rico em detalhes e muito bem escrito, como tudo o que ele se dispõe a fazer. Mas posso dizer que ele superou minhas expectativas e espero que surpreenda você também.

Durante a leitura, você vai aprender que existem diferentes tipos de testes, desde os que testam pequenos pedaços da aplicação, conhecidos como testes unitários, até testes que juntam diferentes pedaços do software e analisam se eles continuam funcionando em

harmonia, que são os testes de integração. Além desses, você também verá alguns não tão falados, mas não menos importantes, como os testes de carga. E não se engane, apesar de técnico, o livro também abre espaço para a prática, onde cada assunto discutido é acompanhado da criação de pequenas e diferentes aplicações, permitindo que você experimente e aprenda praticando junto do livro.

Tenha uma boa leitura e que suas próximas aplicações sejam mais seguras, confiáveis e com a qualidade que só os testes automatizados podem nos dar!

Sobre o autor

Figura -1.1: Gabriel Ramos.

Gabriel Ramos é pintor de pixel, ou desenvolvedor, como algumas pessoas preferem chamar, mentor na Laboratória e instrutor na Caelum.

Já passou por empresas de diversos tamanhos e segmentos: de e-commerces e companhias mais consolidadas a startups unicórnios com produtos emergentes. Na grande maioria das experiências, teve contato com tudo o que envolve o ecossistema JavaScript, desde aplicações front-end a ferramentas e back-end em NodeJS.

Hoje possui uma graduação mais tradicional, mas trabalhou por anos sem uma formação superior e, embora acredite que bases acadêmicas conseguem suprir algumas deficiências de conhecimentos computacionais (principalmente teóricas), é um forte defensor das diversas formas de aprendizagem e tem como

princípio que o estudo, acima de tudo, deve ser encorajado, independente de sua forma.

Mantém um blog pessoal https://gabrieluizramos.com.br/ no qual fala sobre diversos assuntos de tecnologia e, fora isso, também é apaixonado por fotografia, hobby que mantém com muito carinho disponibilizando suas fotos para uso gratuitamente.

Todo mundo na mesma página

A quem se destina este livro

Este livro é para qualquer pessoa que queira se aprofundar em testes ou melhorar seus fundamentos no assunto.

É necessário um conhecimento prévio de JavaScript, porém, se você tem familiaridade com qualquer outra linguagem, não sentirá dificuldades em consumir este conteúdo.

Requisitos e configuração de ambiente local

Para facilitar o entendimento dos assuntos abordados aqui, é interessante que tenhamos um certo ambiente e algumas ferramentas instaladas em nossos computadores. Tudo o que você precisará para acessar e executar os exemplos e instalar os pacotes é basicamente:

- Um navegador de sua escolha;
- Uma IDE ou editor de texto (os projetos serão realizados no Visual Studio Code: https://code.visualstudio.com/);
- O NodeJS (https://nodejs.org/) para executar os códigos necessários e o NPM (https://www.npmjs.com/) para instalar os pacotes e dependências utilizadas, assim como um conhecimento básico dessas duas ferramentas;
- Conhecimento básico de utilização de terminais;
- Conhecimento básico sobre Git (https://git-scm.com/) e versionamento, apenas porque os trechos de códigos serão disponibilizados por meio do GitHub (https://javascriptassertivo.com.br/) e, embora possam ser baixados como ZIP, trabalhar com os repositórios deixará os projetos mais reais.

Embora o livro aborde JavaScript, o foco será o desenvolvimento de testes e não as funcionalidades da linguagem por si só.

Sobre o projeto e as aplicações desenvolvidas

Realizaremos os testes em cima de um grande projeto com um objetivo único: permitir operações de `CRUD` (ou seja, `criar`, `ler`, `atualizar` e `deletar`) para usuários em um arquivo simulando uma base de dados.

Além disso, ao longo dos capítulos, trabalharemos com várias camadas diferentes de aplicações, para que possamos abordar diversas peculiaridades e aspectos dos mais variados tipos de testes.

Inicialmente começaremos os testes por uma aplicação que só é executada através de linha de comando (mais conhecida como CLI). Colocaremos em prática e aprenderemos de forma sólida vários conceitos de testes unitários testando essa ferramenta.

Após isso, vamos desenvolver os testes de uma aplicação back-end utilizando Node e Express que expõe uma API para justamente aplicar as operações de `CRUD` da CLI, que foi citada anteriormente. Nesta etapa, realizaremos os testes unitários e integrados dessa camada da aplicação e finalizaremos com alguns testes de carga.

Depois, vamos focar nos testes da nossa aplicação front-end, que será responsável por entregar uma interface de usuário que se comunicará com a aplicação back-end desenvolvida anteriormente. Testaremos alguns códigos genéricos de navegadores e também especificidades de um framework muito utilizado hoje em dia, o React. Também realizaremos testes unitários e de integração e finalizaremos aplicando os testes de regressão visual em nossos componentes de interface.

Por fim, mas não menos importante, realizaremos nosso teste de ponta a ponta (ou end-to-end/e2e), onde vamos simular um usuário utilizando um navegador, que vai interagir com nossa aplicação front-end e simular um fluxo completo, conforme o esperado.

Como nosso objetivo é aprender e fundamentar exclusivamente os testes, não focaremos na construção das aplicações em si, apenas passaremos pelas partes necessárias para que elas funcionem e também pelas configurações de testes de cada projeto. Claro que, conforme vamos progredindo e desenvolvendo esses testes, conheceremos as aplicações e entenderemos as responsabilidades de cada pedaço delas para que possamos realizar os testes de forma clara, sempre mantendo estes como nosso objetivo principal.

Parte 1: Fundamentos de testes

Ao longo desta primeira parte, aprenderemos toda a teoria e os fundamentos de testes. Conversaremos sobre os motivos pelos quais você deve testar e garantir a confiança do código que você escreve.

Também aproveitaremos para entender mais sobre a estrutura do Jest, o framework de testes que adotaremos ao longo da nossa jornada pelos próximos capítulos.

Capítulo 1
Uma conversa (nem tão) séria sobre testes

1.1 Alguns dos (vários) motivos pelos quais você deveria fazer testes

Talvez você tenha chegado até este livro sem de fato entender os ganhos que testes podem trazer para você, para o seu código, seu produto e até mesmo para sua empresa. Embora não seja um tópico tão novo, a conversa sobre testes (principalmente no universo JavaScript e no Brasil) começou a ter a devida atenção só nos últimos anos.

Após investir tempo escrevendo testes e garantindo qualidade em aplicações, os ganhos ficam mais claros. De todo modo, vamos discutir aqui os principais pontos.

Confiança

Vamos imaginar um cenário completamente controlado para que o exemplo seja realmente claro. Imagine que você faz parte de uma equipe que cuida de um produto, cujo foco é garantir e gerar folhas

de pagamento em um sistema interno de uma empresa. É um exemplo bem simples, mas que pode nos mostrar os benefícios que testes podem nos trazer.

Pelo fato de ter um número razoável de funcionários e por acharem que é perda de tempo fazer o time de desenvolvimento escrever testes em vez de se dedicar às novas funcionalidades, o código-fonte das aplicações dessa empresa não é coberto por teste algum.

Certo dia, você encontra a função que soma um salário final contando horas extras. Algo como:

```
const somaHorasExtas = (salario, valorHorasExtras) => {
    return salario + valorHorasExtras;
};
```

Então, no meio do mês, alguma pessoa desavisada e sem a devida atenção alterou o sinal de `+` por `-`. E o código final ficou mais ou menos assim:

```
const somaHorasExtas = (salario, valorHorasExtras) => {
    return salario - valorHorasExtras;
};
```

A função `somaHorasExtras` é bem importante para o funcionamento do sistema como um todo, mas não é um código executado diariamente. Somente nas datas de pagamento é que ele é, de fato, utilizado e uma mudança dessas poderia facilmente passar despercebida em um ambiente onde a cultura de testes não é formada.

Eis então que o dia de gerar as folhas de pagamento chega. O sistema executa sua funcionalidade perfeitamente, até o momento em que alguma pessoa da empresa nota o erro na sua folha de pagamento.

Essa pessoa muito provavelmente entrará em contato com o pessoal da área de pagamentos (que utiliza o sistema), que, por sua vez, vai buscar os times responsáveis pela plataforma. O pedido

chega até a devida equipe, que começa a procurar incessantemente a alteração e o motivo pelo qual esse *bug* está acontecendo.

Depois de horas (senão dias), você consegue encontrar a causa raiz. Essa simples alteração, que não teve a devida atenção e por não estar coberta por testes, não indicou qualquer possível problema no sistema que estava rodando.

Através dos testes é possível ter diversas garantias de que nosso código funciona como o esperado e teremos vários indicativos caso isso não aconteça. Com isso, podemos ter em mente que a **confiança** é um dos primeiros e principais valores que testes trazem para o código que você escreve.

Qualidade

Quando você possui regras de testes (que aprenderemos a fundo no decorrer deste livro), fica muito evidente a **qualidade** que o seu produto e o seu código entregam.

Além de **confiar** em que seu código funciona como deveria, você também garante exatamente o que ele **não deve fazer**, facilitando e contribuindo para analisar e colher métricas de qualidade inclusive entre times.

É algo um pouco abstrato, mas, quando uma cultura de teste está estabelecida em um ambiente, times tornam-se mais autônomos, e produzir softwares em equipes completamente desacopladas acaba virando uma tarefa trivial.

Cada time é responsável por garantir a qualidade de seus componentes e serviços e, com a devida cultura de testes, o atrito entre as equipes é drasticamente minimizado. A menos que um produto não esteja funcionando como deveria ou não tenham implementado todos os casos de uso necessários para seus clientes, a comunicação entre as equipes pode focar em aprimorar os produtos já existentes.

De certa forma, os times ficam mais independentes e focados em produzir seus produtos com a qualidade que deveriam, pois assim não precisam se preocupar em testar se o restante dos serviços funciona como deveria.

Tempo

Vamos voltar ao exemplo do sistema de folha de pagamento. Já parou para imaginar quantas pessoas (e quantas horas) poderiam ter sido "perdidas" para encontrar a causa raiz do *bug* que estava no sistema?

É comum que muitas pessoas olhem para testes e torçam o nariz, não vejam valor e pensem: "Não vou perder meu tempo escrevendo mais código, confio em que o que eu fiz até aqui está funcionando", afinal, quase ninguém quer dar o braço a torcer e afirmar que comete erros ou que não previu algum cenário, certo?

É por isso que os testes estão muito relacionados com o **ganho de tempo** e não com a **perda**. Tempo em testes é **investido** (e não **perdido**) e traz resultados extraordinários quando pensamos em manter aplicações por um certo tempo.

Só isso?

Se mesmo assim você ainda não se convenceu de que testes trazem muitos benefícios, ao longo dos próximos capítulos, teremos exemplos mais sólidos que farão você repensar a importância deles. Até então, podemos resumir esses pontos em:

> Investir **TEMPO**, garantindo **qualidade** e construindo **confiança** no código que você escreve, com certeza trará bons frutos para o seu produto no longo prazo.

1.2 Como você testa suas aplicações?

Você termina de desenvolver uma aplicação, alinha qualquer detalhe de interface com o time de design e (talvez) faz *deploy* para um ambiente de testes.

Você manualmente começa a navegar pelas telas de sua aplicação com sua nova funcionalidade pronta assumindo papel de um usuário, passando por todos os fluxos para se certificar de que não deixou nenhum detalhe escapar.

Dependendo de onde você trabalha, talvez existam pessoas para ajudar nessa tarefa, sejam elas *QAs*, responsáveis por assegurar a qualidade (*Quality Assurance*) do produto, ou mais pessoas que desenvolvem na sua equipe.

Vocês dividem os fluxos que sua aplicação contém e cada um faz uma parte. Pensando em um cenário de *e-commerce* (comércio eletrônico), vamos imaginar que você criou a funcionalidade de salvar um cartão de crédito na sua conta e reutilizar esse mesmo cartão para finalizar compras futuras. Esse pequeno fluxo pode ser dividido em diversas formas, principalmente quando não existem **testes automatizados** para garantir que as demais formas de pagamento (boleto e/ou cartão de débito) não foram afetadas. Só aqui, já temos ideia de alguns possíveis fluxos e cenários que precisam de atenção:

- Verificar se o cartão está sendo salvo corretamente;
- Verificar se o cartão, após ser salvo, é exibido como forma de pagamento ao final do processo de compra;
- Verificar se a compra é efetuada corretamente ao escolher cartão salvo;
- Verificar se a compra via boleto não foi impactada e se todo o seu fluxo e detalhes permanecem intactos;
- Verificar se a compra via cartão de débito não foi impactada e se todo o seu fluxo e detalhes permanecem intactos;

- Verificar que a compra com um cartão de crédito novo não salva esse determinado cartão caso o usuário não queira.

Só de pensar rapidamente, esses cinco cenários podem preocupar um time que não possui a cultura de testes bem difundida. Vale lembrar que cada fluxo desses cenários pode conter diversas telas, detalhes e campos a serem preenchidos, fazendo com que o trabalho manual de navegar na aplicação seja cada vez maior.

Vamos imaginar "o caminho feliz" ao pensar que os testes manuais foram feitos e que a aplicação está funcionando totalmente como deveria. Na próxima *sprint* (período em que um time de desenvolvimento ágil, geralmente *Scrum*, é focado em desenvolver suas entregas) uma nova funcionalidade será desenvolvida.

Agora, qualquer pessoa que realizar uma compra nesse *e-commerce* poderá ganhar alguns descontos se levar uma certa quantidade de itens de uma mesma categoria. Por exemplo:

- Se uma pessoa coloca dois itens eletrônicos no carrinho, ela deve receber uma mensagem dizendo que, se ela adicionar mais um item da marca X ou Y, ganhará 10% de desconto.
- Se ela adiciona três itens de esporte no carrinho, ela deve saber que, se adicionar mais dois itens da mesma categoria, ganhará 25% de desconto.

De certa forma, essa nova funcionalidade deixa a compra mais lúdica (ou *gamificada*, um termo que se tornou muito popular recentemente) para dar ao usuário a sensação de estar dentro de um jogo enquanto faz compras. Essa funcionalidade tenta instigar essa pessoa a sempre atingir um objetivo, perseguir um próximo nível e desbloquear um novo desconto. Muito provavelmente haveria indicativos na tela do usuário com seu progresso para desbloquear as próximas conquistas.

O time do qual você faz parte desenvolve essa nova funcionalidade e, com isso, chega novamente à etapa de testes. Entretanto, agora

que os cenários já parecem um pouco mais complicados, os testes a serem feitos serão mais ou menos os seguintes:

- Verificar se o fluxo de compra com cartão de crédito não salvo está funcionando sem aplicar o desconto final;
- Verificar se o fluxo de compra com cartão de crédito não salvo está funcionando aplicando o desconto final;
- Verificar se o fluxo de compra com cartão de crédito salvo está funcionando sem aplicar o desconto final;
- Verificar se o fluxo de compra com cartão de crédito salvo está funcionando aplicando o desconto final;
- Verificar se o fluxo de compra com boleto está funcionando sem aplicar o desconto final;
- Verificar se o fluxo de compra com boleto está funcionando aplicando o desconto final;
- Verificar se o fluxo de compra com cartão de débito está funcionando sem aplicar o desconto final;
- Verificar se o fluxo de compra com cartão de débito está funcionando aplicando o desconto final.

Percebeu que a lista aumentou um pouco? Com uma nova funcionalidade, todo o trabalho de realizar testes manuais, infelizmente, precisaria ser refeito, e essa necessidade fica cada vez mais agressiva conforme seu time entrega novos comportamentos nessa aplicação.

Afinal, embora novas funcionalidades sejam atribuídas a fluxos e pagamentos já existentes, todos os testes devem garantir que o sistema e as funcionalidades que já existiam anteriormente continuam funcionando bem, agregando ou não novas opções aos compradores.

Esse foi um exemplo completamente controlado, mas real. Cenários como esse existem em diversos ambientes e vivenciar esses desafios e testes manuais faz com que todos os ganhos que comentamos anteriormente fiquem um pouco mais claros.

Agora podemos começar a entrar em mais alguns detalhes de fundamentos de testes que são extremamente importantes antes de começarmos a colocar a mão na massa e escrever códigos.

1.3 A famosa pirâmide de testes

Se você já se interessava pelo assunto de testes antes deste livro, muito provavelmente você já conhece a imagem que veremos. Caso a esteja vendo pela primeira vez, esta é a **pirâmide de testes**, uma forma de representar algumas categorias básicas de testes com alguns conceitos.

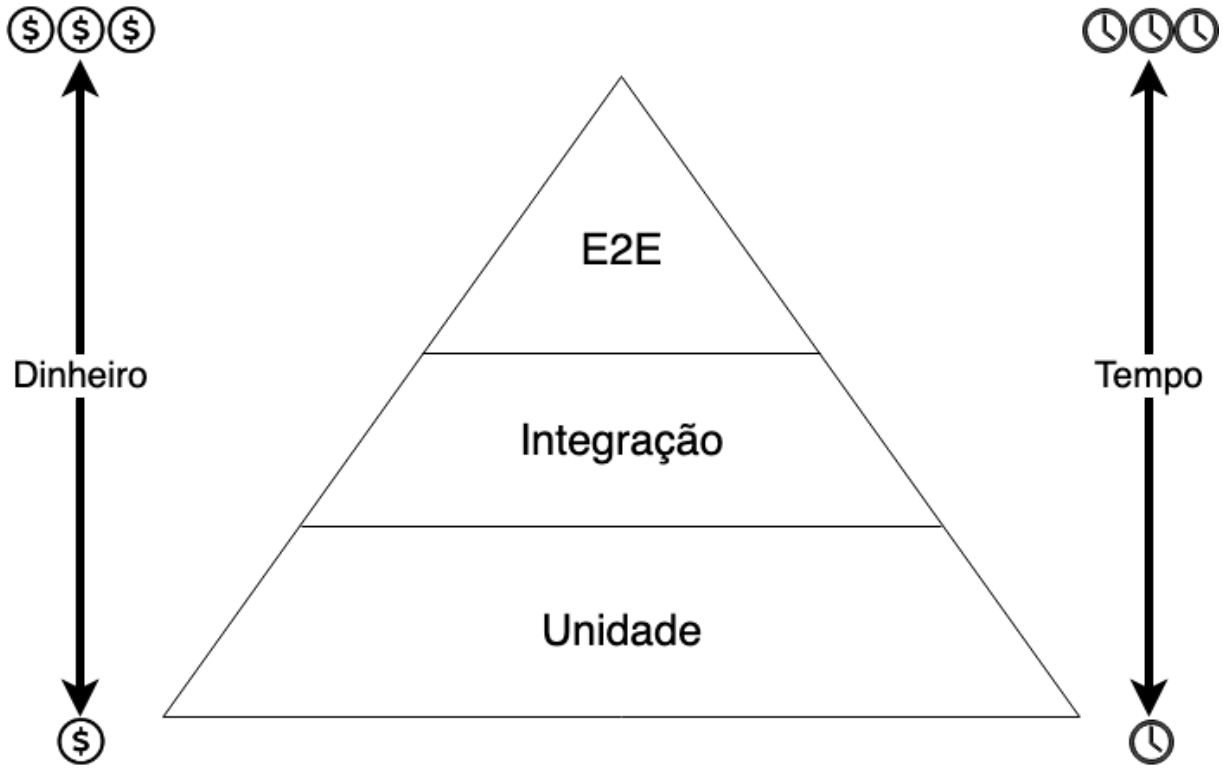


Figura 1.1: Pirâmide de testes.

A imagem contém uma pirâmide dividida em três partes, cada uma indicando uma forma de testar: na sua base, temos escrito "unidade", ao meio, "integração e, no topo, temos escrito "E2E"

(sigla para o termo em inglês *end-to-end*, ou de ponta a ponta, como veremos a seguir). A imagem também contém duas setas bidirecionais, indo do topo à base da pirâmide. A seta da esquerda indica o dinheiro.

Para entender o que a pirâmide de testes quer dizer, vamos analisar a imagem por partes, iniciando pela base da pirâmide.

Unidade

A base da pirâmide indica **unidade**. Pensamos nosso código em pequenos pedaços que são, de fato, pequenas unidades isoladas, sejam elas pequenas funções, utilitários ou componentes de interface (como um pequeno botão que compõe uma tela completa).

Integração

Indicados no meio da pirâmide, são testes um pouco mais completos, que envolvem o teste de várias unidades em conjunto e, por isso, recebem o nome de **integração**. Isso quer dizer que pensamos em testes de integração quando testamos em conjunto diferentes trechos de código (que também poderiam ser testados de forma isolada e unitariamente).

E2E

Sigla para *end-to-end* (de ponta a ponta) e está, geralmente, representada no topo da pirâmide. Indica uma categoria de testes mais robusta, cuja ideia é simular um usuário interagindo com navegadores e, de fato, utilizando o sistema da forma mais real possível.

As setas bidirecionais: dinheiro e tempo

Você já deve ter se cansado de ouvir que tempo é dinheiro e, embora isso seja um pouco parecido, a ideia é olhar esses dois parâmetros de forma diferente.

Testes unitários são mais rápidos de serem feitos, já que quebramos nossos códigos em pedaços menores e garantimos que cada um deles funciona como deveria. Portanto, o investimento de tempo e dinheiro, sejam as horas de uma pessoa desenvolvendo ou até mesmo uma estrutura de *CI* (*Continuous Integration*, ou integração contínua) para rodar esses testes de forma automatizada, é relativamente menor.

Já no caso dos testes de integração, embora muitas vezes as ferramentas sejam as mesmas já utilizadas nos testes de unidade, escrever o teste, em si, é algo um pouco mais demorado e envolve cenários mais completos a serem testados. Embora o custo de ferramentas de integração contínua seja praticamente o mesmo, o investimento em tempo acaba sendo um pouco maior e, pensando no cenário onde pessoas são pagas para desenvolverem código, o preço acaba sendo elevado também.

Por último, mas não menos importante, o teste *E2E*, por envolver uma estrutura completa da aplicação (tanto *front-end*, quanto *back-end*), acaba sendo um teste mais demorado e ainda mais caro, já que agora os fluxos de usuário deverão ser simulados de forma completa de ponta a ponta. Isso inclui um ambiente e uma estrutura de testes mais robustos, mais caros, o que consequentemente demanda mais tempo para escrever o teste e configurar todas as pontas necessárias para que tudo funcione corretamente.

Em outras palavras, podemos pensar que escrever e manter os testes que se aproximam da base da pirâmide é uma tarefa que tende a ser mais rápida e financeiramente mais barata de ser realizada, já os testes que se aproximam do topo acabam sendo um pouco mais demorados e com um preço mais elevado.

Estático

Alguns autores, como o Kent C. Dodds (https://kentcdodds.com/) em seu curso *Testing JavaScript* (https://testingjavascript.com/), consideram a etapa estática como uma outra etapa fundamental dos testes, vindo antes de qualquer uma das que são exibidas nas pirâmides "tradicionais". Trata-se da análise de código antes de realizar qualquer transpilação ou compilação e até mesmo antes de executar.

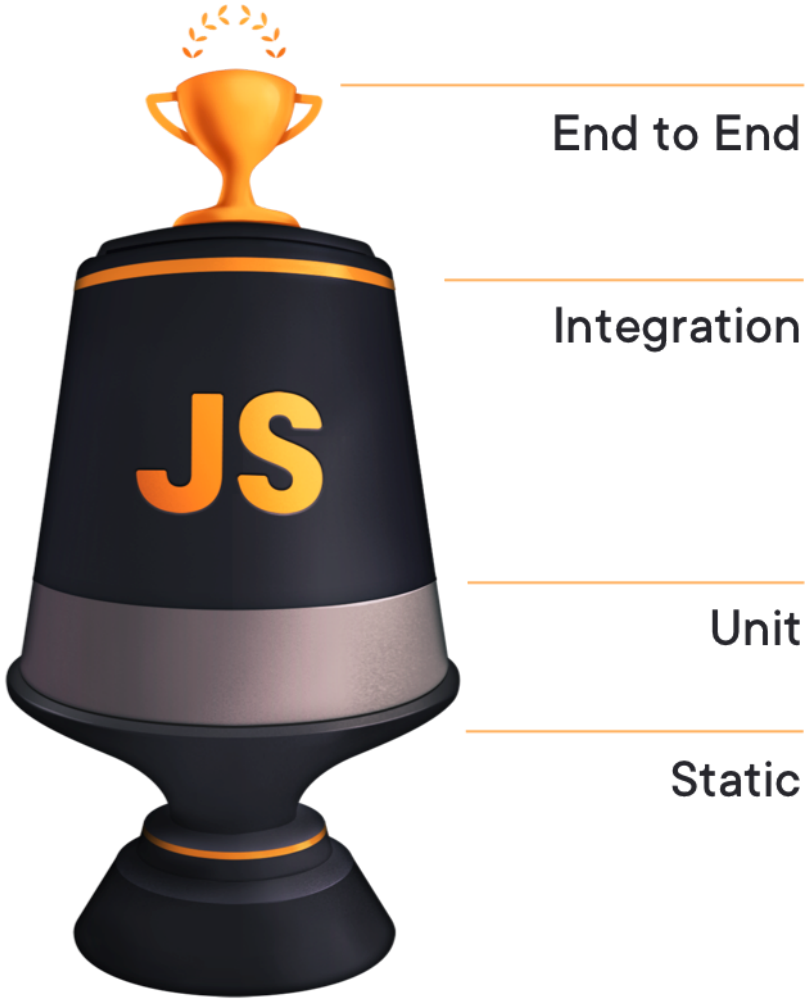


Figura 1.2: Troféu de testes do curso Testing JavaScript.

Particularmente, concordo bastante com esse modelo. Já que a ideia dos testes é criar confiança no código que escrevemos e se podemos fazer isso de uma forma simples e totalmente estática, por que não?

1.4 Esses detalhes são apenas conceitos

Grande parte desses conceitos pode ser facilmente adaptável dependendo do contexto, do produto e da forma como você organiza seus testes. É comum vermos, por exemplo, o topo da pirâmide com a nomenclatura *UI* (que significa *User Interface*, ou interface de usuário), abrindo brechas para diferentes interpretações sobre o que de fato o teste faz.

Alguns outros desenhos colocam no meio da pirâmide o termo "serviço", indicando que é um serviço completo que deve ser testado, mas também uma interpretação muito voltada a cenários de back-end com serviços desacoplados.

O que podemos tirar de fundamentos da pirâmide de testes é que podemos dividir nosso código e garantir qualidade em diversas camadas possíveis, sejam elas diretamente com trechos isolados, com comportamentos de telas e componentes compostos, ou simulando um usuário interagindo com o nosso produto.

Esses conceitos são os fundamentais, já as nomenclaturas são apenas detalhes de cada interpretação.

CAPÍTULO 2
Análise estática de código com ESLint

2.1 Um passo para trás, dois passos para frente

Para iniciar nossa jornada nos testes, vamos começar com a análise estática de código e entender como ela é aplicada.

Podemos imaginar a análise estática como uma etapa de testes que é executada em tempo de desenvolvimento, podendo, inclusive, nos dar feedback enquanto escrevemos nosso código.

Essa etapa é fundamental para garantir que alguns pequenos erros não sejam cometidos antes mesmo de aplicar qualquer tipo de teste, sejam unitários ou integrados.

Lembre-se de que todo o código está disponível para ser clonado. O projeto em que aplicaremos esses primeiros conceitos está no repositório que contém os exemplos do livro (https://javascriptassertivo.com.br/). Trabalharemos em cima do conteúdo da pasta `projetos/01-fundamentos-de-testes`. Começaremos com a pasta `01-eslint`.

Por exemplo: vamos imaginar que declaramos uma variável `idade` e simplesmente somamos `+ 1`.

```
const idade = 24;
const novaIdade = idade + 1;
```

É um código perfeitamente funcional. Se o executarmos em um navegador (através do *DevTools*) ou criarmos um arquivo e o executarmos com o Node, também funciona.

Entretanto, vamos imaginar que, por algum motivo, acabamos escrevendo errado a variável `idade` na segunda linha. Nosso código então poderia ficar algo mais ou menos assim:

```
const idade = 24;
const novaIdade = idad + 1;
```

Esquecemos de colocar a letra `e` ao final da variável `idade` e temos um erro na nossa declaração. Ao executar novamente esse código, veremos a seguinte mensagem de erro:

```
ReferenceError: idad is not defined
```

O que aconteceu foi o seguinte: ao executarmos esse código, o JavaScript criou nossa variável `idade` com o valor `24` e, logo após, tentou criar a variável `novaIdade`. Entretanto, ao chegar a essa etapa, devido ao fato de a variável `idade` estar escrita errado (`idad`) tivemos um erro de referência. Em outras palavras, estávamos tentando utilizar uma variável que não foi criada.

Claro que é um exemplo muito simples, mas é justamente o tipo de erro que ferramentas de análise estática podem nos ajudar a identificar. Elas nos poupam de todo o trabalho de testar e executar um trecho de código apenas para pegar esses erros pequenos.

A ferramenta que veremos a seguir, em específico, é um tipo de *linter*. *Linters* são justamente ferramentas que nos ajudam a assegurar que determinado código segue determinada regra e estrutura de escrita (podendo, inclusive, nos ajudar a formatar um código). Dessa forma, podemos ter uma configuração em um projeto e fazer com que todos os arquivos sigam um mesmo padrão de escrita.

Vale lembrar que linters não são ferramentas específicas de JavaScript e podem ser utilizadas em praticamente qualquer linguagem.

No nosso caso, utilizaremos um linter chamado **ESLint** (https://eslint.org/), que é um dos mais utilizados no ecossistema JavaScript.

2.2 Instalação

Para instalar o ESLint, tudo o que precisaremos fazer é abrir o terminal na pasta do projeto onde queremos instalá-lo. Esse projeto já deve ter uma estrutura para trabalhar com Node: um arquivo `package.json` contendo as dependências e configurações; caso você não tenha, basta executar o seguinte comando no seu terminal para criar essa estrutura:

```
npm init -y
```

Esse comando criará o arquivo `package.json` com alguns dados já pré-configurados. Realizar esse processo será necessário para praticamente todos os códigos que desenvolveremos ao longo do livro.

Para instalar o ESLint, basta instalá-lo como dependência de desenvolvimento no nosso projeto utilizando o NPM.

```
npm install --save-dev eslint
# ou, de maneira mais curta
npm i -D eslint
```

A partir de agora, o ESLint já está instalado no nosso projeto, no entanto precisamos configurá-lo para que ele saiba as regras que deve ou não aplicar nos nossos arquivos.

2.3 Configuração

Para iniciar a nossa configuração, podemos utilizar o NPX (o executável de pacotes dos NPM). Já temos essa ferramenta por padrão, pois ela já é instalada junto do Node e do NPM. Para utilizá-la, não tem mistério: basta executarmos `npx` e informarmos o pacote que queremos executar e também passar qualquer argumento extra.

No nosso caso, queremos executar o pacote do ESLint que instalamos e passar o argumento `--init`, informando que vamos iniciar nossa configuração. Agora, no terminal, basta executar:

```
npx eslint --init
```

Ao executar esse comando no terminal, uma tela aparecerá para nós com a seguinte mensagem:

```
? How would you like to use ESLint? …
  To check syntax only
? To check syntax and find problems
  To check syntax, find problems, and enforce code style
```

O que essa mensagem está nos perguntando é "Como você gostaria de utilizar o ESLint?", e as três opções são:

- Para checar apenas sintaxe.
- Para checar sintaxe e encontrar problemas.
- Para checar sintaxe, encontrar problemas e forçar estilo de código.

Podemos selecionar uma opção utilizando as setas do teclado e apertando a tecla `enter`. Para aceitarmos a opção mais restritiva de todas, vamos selecionar a última. Agora temos uma nova pergunta:

```
? How would you like to use ESLint? · style
? What type of modules does your project use? …
? JavaScript modules (import/export)
  CommonJS (require/exports)
  None of these
```

O que está sendo questionado é "Que tipo de módulos seu projeto utiliza?", e as três opções são:

- Módulos JavaScript (import/export).
- CommonJS (require/exports).
- Nenhum desses.

Essa pergunta diz respeito à forma como nossos arquivos lidarão com módulos. Caso você queria saber mais sobre esse assunto, em específico, o post em https://gabrieluizramos.com.br/modulos-em-javascript explica detalhadamente a história de módulos na linguagem JavaScript.

Como nosso exemplo ainda é pequeno, não precisamos nos preocupar com isso neste momento. Basta selecionar a primeira opção e mais uma pergunta aparecerá:

```
? How would you like to use ESLint? · style
? What type of modules does your project use? · esm
? Which framework does your project use? …
? React
  Vue.js
  None of these
```

O que nos está sendo perguntado é "Qual framework seu projeto usa?", e temos as três opções:

- React
- Vue.js
- Nenhum desses

Como nosso projeto não utilizará nenhum desses frameworks (por enquanto), selecionaremos a última opção. Outra pergunta aparecerá:

```
? How would you like to use ESLint? · style
? What type of modules does your project use? · esm
? Which framework does your project use? · none
? Does your project use TypeScript? › No / Yes
```

Dessa vez, a pergunta é "Seu projeto usa TypeScript?", e temos apenas duas opções na mesma linha: "não" ou "sim". Podemos selecionar a primeira opção, pois não usaremos TypeScript.

Agora a pergunta é:

```
? How would you like to use ESLint? · style
? What type of modules does your project use? · esm
? Which framework does your project use? · none
? Does your project use TypeScript? · No / Yes
? Where does your code run? …  (Press <space> to select, <a> to toggle
all, <i> to invert selection)
? Browser
? Node
```

Essa pergunta é um pouco mais elaborada e nos questiona "Onde seu código roda? (Aperte `<espaço>` para selecionar, `<a>` para alternar todos, `<i>` para inverter a seleção)", e temos como opções o Navegador e o Node. Como isso ainda não nos interessa muito, podemos selecionar todos apertando `a` .

Mais uma pergunta:

```
? How would you like to use ESLint? · style
? What type of modules does your project use? · esm
? Which framework does your project use? · none
? Does your project use TypeScript? · No / Yes
? Where does your code run? · browser, node
? How would you like to define a style for your project? …
? Use a popular style guide
  Answer questions about your style
  Inspect your JavaScript file(s)
```

Dessa vez a pergunta é "Como você gostaria de definir um estilo para o seu projeto?", e temos as seguintes opções:

- Usar um guia de estilo popular.
- Realizar perguntas sobre seu estilo.
- Inspecionar seu(s) arquivo(s) JavaScript.

Para facilitar nosso processo de configuração, vamos selecionar a primeira opção. E a pergunta da vez é:

```
? How would you like to use ESLint? · style
? What type of modules does your project use? · esm
? Which framework does your project use? · none
? Does your project use TypeScript? · No / Yes
```

```
? Where does your code run? · browser, node
? How would you like to define a style for your project? · guide
? Which style guide do you want to follow? …
? Airbnb: https://github.com/airbnb/javascript
  Standard: https://github.com/standard/standard
  Google: https://github.com/google/eslint-config-google
```

Isto é "Qual guia de estilo você quer seguir?", e as opções são:

- Airbnb: https://github.com/airbnb/javascript.
- Padrão: https://github.com/standard/standard.
- Google: https://github.com/google/eslint-config-google.

Cada uma dessas opções é seguida de um link, com os arquivos das regras de ESLint indicadas. Empresas, como Airbnb e Google, deixam seus padrões públicos para que a comunidade possa se aproveitar desse conteúdo. Vamos selecionar a primeira opção.

A pergunta agora é (estamos terminando, juro!):

```
? How would you like to use ESLint? · style
? What type of modules does your project use? · esm
? Which framework does your project use? · none
? Does your project use TypeScript? · No / Yes
? Where does your code run? · browser, node
? How would you like to define a style for your project? · guide
? Which style guide do you want to follow? · airbnb
? What format do you want your config file to be in? …
? JavaScript
  YAML
  JSON
```

"Em qual formato você quer que esteja seu arquivo de configuração?", e temos três opções:

- JavaScript.
- YAML.
- JSON.

Qualquer uma das opções funcionaria bem. Vamos selecionar a última. Após finalizarmos essa etapa, o ESLint instalará os pacotes que selecionamos na etapa do guia de estilo (no nosso caso, o do Airbnb)

```
Checking peerDependencies of eslint-config-airbnb-base@latest
The config that you've selected requires the following dependencies:
```

E nos perguntará se queremos instalar os pacotes de configuração:

```
eslint-config-airbnb-base@latest eslint@^5.16.0 || ^6.8.0 || ^7.2.0
eslint-plugin-import@^2.21.2
? Would you like to install them now with npm? › No / Yes
```

"Você gostaria de instalar os pacotes agora com o NPM?", e temos as opções "não" e "sim". Vamos selecionar a última opção indicando que queremos instalar os pacotes.

Então, o NPM instalará os pacotes para nós e, ao fim do processo, teremos a seguinte mensagem:

```
Successfully created .eslintrc.json file in /caminho-do-seu-projeto
```

E pronto, já finalizamos nossa configuração inicial. Isso criará um arquivo `.eslintrc.json` na nossa pasta (veremos com cuidado esse arquivo em breve).

Vamos fazer mais um pequeno ajuste. No arquivo `package.json`, criaremos um script para rodar o ESLint e verificar se nossos arquivos estão de acordo com as nossas regras de configuração.

No arquivo `package.json`, na parte de `scripts`, basta adicionar um código que executa o ESLint utilizando o NPX e informando a pasta que queremos verificar, da seguinte forma:

```
"scripts": {
  "lint": "npx eslint ./pasta-dos-seus-arquivos", // inserimos esta linha
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

Caso seus arquivos estejam na mesma pasta onde você criou o arquivo de configuração, basta trocar a `./pasta-dos-seus-arquivos` no script por um ponto final `.`, o que fará com que o ESLint verifique todos os arquivos no diretório atual.

Agora basta criarmos um arquivo nesse projeto com aquele trecho de código que usamos de exemplo

```
const idade = 24;
const novaIdade = idad + 1;
```

E executarmos o comando `lint`, que criamos, da seguinte forma:

```
npm run lint
```

Após realizar essa etapa, já conseguiremos perceber alguns erros:

```
1:7   error  'idade' is assigned a value but never used      no-unused-
vars
2:7   error  'novaIdade' is assigned a value but never used  no-unused-
vars
2:19  error  'idad' is not defined                           no-undef
2:28  error  Newline required at end of file but not found   eol-last

? 4 problems (4 errors, 0 warnings)
  1 error and 0 warnings potentially fixable with the `--fix` option.
```

Com isso, já temos a nossa validação funcionando como deveria! Cada linha de erro é composta por alguns blocos de informações, como:

- A `linha:coluna` do erro: como podemos ver no primeiro erro, ele está sendo disparado pela linha 1 e coluna 7.
- O tipo do erro: se é um erro ou apenas um alerta (veremos como configurar isso em breve).
- A mensagem de erro: qual problema o ESLint encontrou e nos informa que deve ser corrigido.
- A regra de erro que foi disparada (como `no-unused-vars`): indica a qual regra o erro pertence (também veremos como ajustar isso em breve).

Ao final dessas linhas, temos um relatório contando quantos problemas e quantos alertas foram encontrados. Com isso em mente, podemos resumir que temos os seguintes erros atualmente:

- erro: variável `idade` teve um valor atribuído, mas nunca foi utilizada.
- erro: variável `novaIdade` teve um valor atribuído, mas nunca foi utilizada.
- erro: variável `idad` não está definida.
- erro: nova linha necessária ao final do arquivo, mas não foi encontrada.

Vamos manter esses erros em mente pois vamos testá-los e modificá-los a seguir.

Ajustando nosso editor para nos ajudar nessa tarefa

Nosso próprio editor pode nos ajudar nessa tarefa também, basta adicionar um plugin do ESLint. No caso do Visual Studio Code, o plugin é este: https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint. Outros editores e IDEs também devem ter seus próprios plugins.

Ao instalá-lo, também já teremos esses erros diretamente ao editar nossos arquivos. Vamos abrir o arquivo que criamos.

```
const idade = 24;
const novaIdade = idad + 1;
```

Figura 2.1: Erro geral no editor.

Parece preocupante que algumas linhas vermelhas apareceram no nosso código, mas não se preocupe, isso é o próprio ESLint já entrando em ação enquanto editamos nosso arquivo, exatamente como queríamos!

Para verificarmos os erros, podemos passar o mouse sobre essas linhas vermelhas, que conseguiremos ver o que está acontecendo. Vamos passar o mouse em cima da variável `idad`, que escrevemos errado.

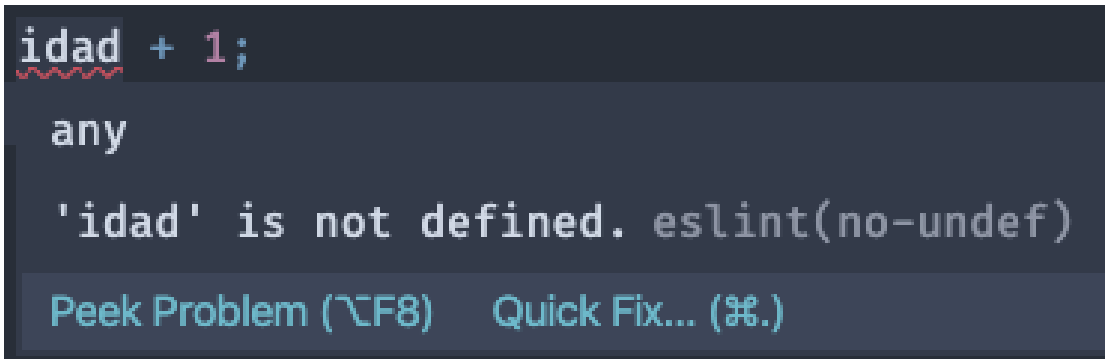


Figura 2.2: Mensagem de erro no editor.

O que a mensagem está nos informando é justamente que a variável `idad` não foi definida. Praticamente o mesmo erro que pegaríamos no nosso navegador. Se quisermos corrigi-lo, basta ajustar o nome da variável que o alerta sumirá.

2.4 Arquivo de configuração, regras e plugins

Para que o ESLint funcione corretamente, ele se baseia em um arquivo de configuração (aquele arquivo em extensão JSON que selecionamos em determinada etapa das perguntas que respondemos anteriormente).

Você deve ter notado que agora temos um novo arquivo no projeto chamado `.eslintrc.json`. Geralmente, arquivos com sufixo `rc` são utilizados para especificar arquivos de configurações para plugins e comandos que podemos executar em um projeto.

Vamos dar uma olhada nesse arquivo mais a fundo:

```
{
    "env": {
        "browser": true,
        "es2021": true,
        "node": true
    },
    "extends": [
        "airbnb-base"
    ],
    "parserOptions": {
        "ecmaVersion": 12,
        "sourceType": "module"
    },
    "rules": {
    }
}
```

Todas essas informações condizem com as respostas das várias perguntas que respondemos anteriormente. Vamos entender o que cada uma delas significa, pois temos bastante coisa nova aqui.

env

Objeto que indica quais ambientes o ESLint validará (para que algumas variáveis globais possam ser analisadas corretamente). Dentro desse objeto, temos as seguintes opções:

- `browser` : códigos que serão executados no navegador (variáveis globais como `window`).
- `es2021` : alguns métodos mais modernos de JS.
- `node` : códigos que serão executados em Node (variáveis globais como `require`).

Se quiser ver o que acontece ao modificar esses valores em `env` , faça um teste removendo a linha `browser` , desabilitando essa opção ou marcando-a como `false` :

```
"browser": false,
```

No seu código, tente acessar alguma propriedade de `window`, como `fetch`, e você verá o seguinte:

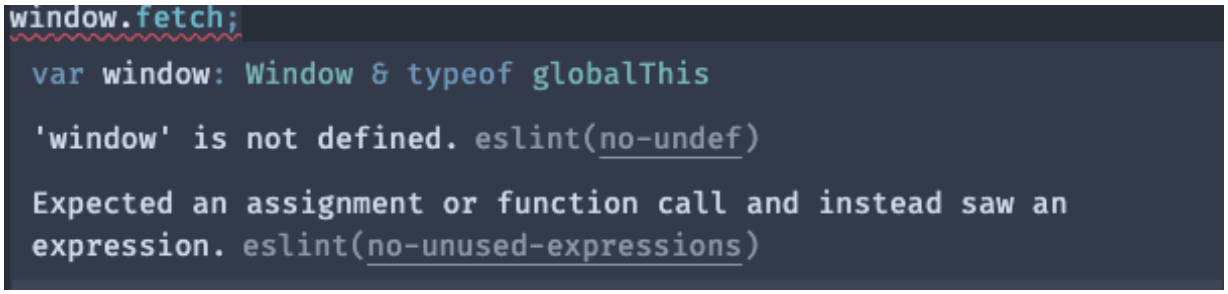

window.fetch;
var window: Window & typeof globalThis
'window' is not defined. eslint(no-undef)
Expected an assignment or function call and instead saw an expression. eslint(no-unused-expressions)

Figura 2.3: Mensagem de erro por alteração de ambiente.

Caso você rode o ESLint pelo comando que criamos (`npm run lint`) o mesmo erro aparecerá.

extends

Array de opções utilizado para estender regras já predefinidas e plugins do ESLint. No nosso caso, como optamos por utilizar as regras do Airbnb como referência, esse é o valor indicado lá.

- `airbnb-base` : regras bases de lint do Airbnb.

Se também quiser testar e ver o que acontece ao modificar essa opção, remova a string `"airbnb-base"`, deixando a opção `extends` vazia:

```
"extends": [
],
```

E rode novamente o comando do lint ou abra o editor e você verá que todos os erros sumiram:

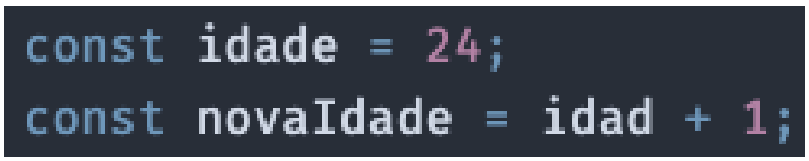


Figura 2.4: Sem mensagens de erro por alterar o extend.

parserOptions

Objeto que contém algumas opções da linguagem que o ESLint deve ou não suportar enquanto analisa o código.

- `ecmaVersion` : versão da especificação ECMAScript (que compõe a linguagem JavaScript);
- `sourceType` : tipo de módulo que seu projeto estará utilizando.

rules

Provavelmente uma das opções mais importantes e aquela na qual você mais investirá tempo de configuração no seu dia a dia. É um objeto de configuração que indica quais regras devem ser habilitadas, tratadas como alerta ou como erro.

Para que possamos entender como isso funciona, vamos adicionar a seguinte linha dentro desse bloco `rules` :

```
"rules": {
    "no-unused-vars": "off"
}
```

O que essa linha faz é desabilitar a regra de variáveis não utilizadas, que é definida pelo `airbnb-base` . Se voltarmos ao editor (ou rodarmos o comando de lint), perceberemos que alguns dos erros anteriores serão omitidos:

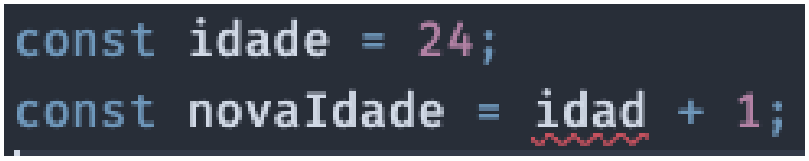


Figura 2.5: Alguns erros sumiram.

Agora só temos o erro na variável `idad` . O erro "variável não utilizada" nas variáveis `novaIdade` e `idade` não existem mais.

Podemos configurar essas regras para estarem habilitadas (exibindo apenas um alerta ou um erro), desabilitadas e até customizá-las

com parâmetros extras. Vamos fazer uma alteração e habilitar a regra apenas para nos exibir um alerta. No arquivo `.eslintrc.json`, faça a seguinte alteração:

```
"rules": {
    "no-unused-vars": "warn"
}
```

As cores desses alertas mudaram. Estão indicados em amarelo:

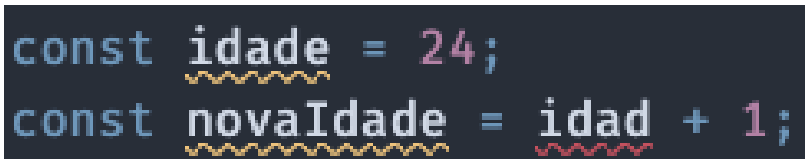


Figura 2.6: Alguns erros viraram alertas (em amarelo).

Se rodarmos o comando `npm run lint`, também veremos essas mensagens apenas como alertas:

```
1:7   warning  'idade' is assigned a value but never used      no-unused-
vars
2:7   warning  'novaIdade' is assigned a value but never used  no-unused-
vars
2:19  error    'idad' is not defined                           no-undef
```

Podemos ajustar essas configurações de erro/alerta/desabilitado usando os valores:

- `"off"` ou `0`: para desabilitar uma regra;
- `"warn"` ou `1`: para habilitar como alerta;
- `"error"` ou `2`: para habilitar como erro.

Não esqueça de remover todos os testes que foram feitos e deixar o arquivo `.eslintrc.json` do jeito que estava, para que você possa ter uma base caso precise consultá-lo. Caso precise, mais acima temos um exemplo do arquivo inicial também.

E mais...

Essas foram algumas das configurações principais do ESLint e são as mais utilizadas na grande maioria dos projetos.

Existem algumas opções extras que podem ser habilitadas e configuradas. Caso você precise, a documentação do ESLint tem uma página específica só sobre essas configurações: https://eslint.org/docs/user-guide/configuring.

2.5 Alguns passos extras com análise estática

Criando sua própria regra de ESLint

Talvez você precise criar uma regra customizada para que o ESLint valide seu código de acordo com alguma definição sua ou da equipe na qual você trabalha, mas, como essa não é uma tarefa tão cotidiana, acho que não vale a pena usar nosso tempo para aprender a criá-la agora, principalmente porque demanda alguns conhecimentos extras sobre *AST* (árvore de sintaxe abstrata, ou *abstract syntax tree*) e algumas coisas mais profundas sobre JS que podem ser estudadas à parte mediante necessidade.

Caso você precise ou tenha curiosidade em se aprofundar, o post *Escrevendo suas próprias regras lint* (https://gabrieluizramos.com.br/escrevendo-suas-proprias-regras-de-lint), que escrevi há algum tempo no meu blog, tem detalhadamente todo o passo a passo para criar um plugin e suas regras customizadas de ESLint.

Prettier

Prettier (https://prettier.io/) é um formatador de código que pode ser utilizado em conjunto com o ESLint para assegurar que algumas regras de estilo de código estão sendo aplicadas como deveriam.

Quando configurados em conjunto, Prettier e ESLint formam uma dupla bem poderosa. É possível utilizar um para formatação de código e outro apenas para validação e prevenção de pequenos *bugs*. Sua forma de utilização é bem parecida com a do ESLint.

- Precisará ser instalado via NPM no projeto;
- Configurado com um arquivo parecido com o que configuramos para o ESLint (podendo até se chamar `.prettierrc.json` também);
- Poderá ser executado via linha de comando (como fizemos o comando `lint` no projeto);

Também é possível instalar plugins nos editores e IDEs para utilizar as formatações do Prettier mais facilmente, e conseguimos configurar o ESLint para interpretar algumas regras do Prettier, caso seja necessário.

Caso queira dar um passo além, você também pode instalar e configurar o Prettier no seu projeto.

Editorconfig

Editorconfig (https://editorconfig.org/) é um arquivo de configuração para que os editores e IDEs possam facilitar algumas visualizações e questões de espaçamento (como tabs ou espaços de indentação).

Com o crescimento do Prettier, citado anteriormente, e sua formatação automatizada na comunidade, muitos projetos acabaram deixando de lado o Editorconfig. Pode ser que seu projeto nem o utilize, ainda assim, por ter sido uma ferramenta utilizada por tanto tempo, acho que vale a pena trazê-la aqui também.

Para começar a usar o Editorconfig é bem simples:

- Você precisa instalar o plugin com suporte ao Editorconfig para seu editor/IDE funcionar corretamente;
- Você precisa criar um arquivo `.editorconfig` com as regras que deseja configurar (contendo os espaçamentos, indentações e

configurações extras).

E pronto, a partir de agora você já deve ter alguns guias visuais mais estruturados, seguindo sua configuração. Por se tratar de uma ferramenta simples de se aplicar e utilizar, eu não vejo motivo para não a utilizar.

TypeScript

Entramos em um outro território diferente aqui, mas que vale a pena ser citado. Quando falamos de análise estática de código, podemos incluir TypeScript (https://www.typescriptlang.org/) na conversa também, mesmo que seu papel seja um pouco maior do que isso.

Embora o TS (sigla para TypeScript) traga várias outras funcionalidades e, principalmente, o "benefício" da tipagem estática enquanto escrevemos código JavaScript, vale ter em mente que esta é mais uma ferramenta intermediária a ser inserida no seu processo de desenvolvimento, sendo necessário instalá-la, configurá-la, aprender como ela funciona (já que é uma ferramenta bem mais robusta) e tudo mais, pelo menos nas aplicações JavaScript mais tradicionais.

Pelo fato de o TS ser uma camada em cima do JS (ou *superset*) e adicionar essas novas funcionalidades, boa parte delas não está presente (e muito provavelmente nem estarão no futuro) na linguagem JavaScript por si só. Isso exige todo um trabalho de configuração de *build* (etapa que compilará o código TS para JS) e ajuste conforme as necessidades do projeto. Isso é necessário na grande maioria das aplicações tradicionais que rodarão em um navegador ou em Node e que são escritas com TypeScript, sem contemplar o Deno (https://deno.land/), o *runtime* que permite a execução de código TS lançado recentemente.

Próxima parada: frameworks de teste

Agora que entendemos como aplicar uma ferramenta de análise estática e algumas outras que podem nos auxiliar em nosso ambiente de desenvolvimento, podemos começar a olhar como alguns frameworks de teste funcionam.

Vamos aproveitar a oportunidade para escrever alguns testes bem básicos para que possamos entender como eles funcionam e também criar nossa versão de um "miniframework" de testes.

Capítulo 3
Simulando um framework de testes

Para que possamos entender um pouco mais como alguns frameworks de teste funcionam, vamos criar nossa pequena versão do Jest, ferramenta que utilizaremos em grande parte dos próximos capítulos.

O código deste capítulo corresponde à pasta `projetos/01-fundamentos-de-testes/02-frameworks-e-ambientes` no repositório https://javascriptassertivo.com.br/, que contém todo o código dos projetos que veremos aqui.

Com isso, vamos começar a falar um pouco mais de código. Lembra do exemplo que comentamos no primeiro capítulo, sobre a função de cálculo de salário?

```
const somaHorasExtras = (salario, valorHorasExtras) => {
    return salario + valorHorasExtras;
};
```

Chegou a hora de escrever um teste bem básico para ela e garantir que ela está funcionando como deveria. Como é uma função bem pequena, podemos utilizá-la para entender como um framework de testes vai se comportar. Afinal, não vamos ficar escrevendo todas essas ferramentas na mão, mas é muito válido entender como elas funcionam.

3.1 O que é, do que se alimentam e como se executam testes?

Você já deve ter tentando imaginar como os testes serão feitos, certo?

Da mesma forma que temos uma camada para o `HTML`, para o `CSS` e para o `JS`, teremos uma nova camada (com mais código) responsável por garantir que nosso código atual está funcionando como deveria.

Se quiséssemos validar que nossa função de soma está funcionando corretamente utilizando JavaScript, como você faria? Tire alguns segundos para imaginar antes de prosseguir. Esse pensamento de refletir o comportamento de nossas funções e aplicações é uma das partes mais fundamentais ao escrever testes.

Se você pensou em fazer alguma validação com `if/else`, verificando se o valor está correto, você acertou! Blocos de `if/else` são as estruturas mais básicas de verificação que temos na linguagem.

No mesmo arquivo da nossa função `somaHorasExtras`, vamos começar a estruturar nosso código então. Vamos criar nossa estrutura de validação:

```
const somaHorasExtras = (salario, valorHorasExtras) => {
    return salario + valorHorasExtras;
};

if () {
// validação se o teste passar
} else {
// validação se o teste falhar
}
```

Perfeito, temos um bloco de `if/else`. Agora vamos criar uma variável para guardar algum valor que esperamos como retorno da função:

```
const esperado = 10;

if () {
// validação se o teste passar
} else {
```

```
// validação se o teste falhar
}
```

Vamos executar nossa função para validar que o retorno dela é igual ao que esperamos. Pela implementação da função `somaHorasExtras`, podemos identificar que, para chegarmos ao resultado `10`, precisaremos passar `5` e `5` como parâmetros.

Então vamos criar uma variável com esses valores:

```
const esperado = 10;
const retorno = somaHorasExtras(5, 5);

if () {
// validação se o teste passar
} else {
// validação se o teste falhar
}
```

Já temos nossa estrutura de validação e nossas variáveis responsáveis por armazenar os valores `esperado` e o `retornado` pela função. Tudo o que precisamos fazer agora é verificar se ambos os valores são iguais. Podemos criar duas mensagens customizadas, uma para caso o valor esteja correto e outra para caso o valor esteja errado, mais ou menos assim:

```
const esperado = 10;
const retorno = somaHorasExtras(5, 5);

if (esperado === retorno) {
    console.log(`? Teste passou`);
} else {
    console.error(`?? Ih, deu ruim...`);
}
```

Com tudo isso pronto, nosso arquivo completo deve ser parecido com isso:

```
const somaHorasExtras = (salario, valorHorasExtras) => {
    return salario + valorHorasExtras;
};
```

```
const esperado = 10;
const retorno = somaHorasExtras(5, 5);

if (esperado === retorno) {
    console.log(`? Teste passou`);
} else {
    console.error(`?? Ih, deu ruim...`);
}
```

Para testar esse arquivo via Node, basta executar no seu terminal:

```
node nome-do-seu-arquivo.js
```

Caso prefira executar no seu navegador, basta colar o código acima na aba `Console` no seu `DevTools`. O ideal é começarmos a focar nos testes com Node, pois é utilizando seus comandos que vamos configurar e deixar tudo automatizado para nós, ok?

Como nosso código está perfeitamente funcional, você deve receber a seguinte mensagem:

```
? Teste passou
```

Com isso, **você acaba de escrever seu primeiro teste**! Exatamente isso, testes são, por trás de qualquer framework e abstração, sequências de validações verificando se o código que escrevemos está funcionando como deveria.

Para nos certificarmos de que ele está funcionando perfeitamente, vamos mudar o sinal de `+` da função para `-`, gerando aquela quebra indesejada, só para vermos o teste nos alertando do erro:

```
const somaHorasExtras = (salario, valorHorasExtras) => {
    return salario - valorHorasExtras; // modificamos + para -
};

// todo o teste continua igual
```

E vamos rodar o teste novamente:

```
?? Ih, deu ruim...
```

Perfeito! Temos segurança de que está tudo como deveria.

3.2 Criando novas abstrações e iniciando nosso miniframework

Agora que entendemos como nosso teste funciona, vamos imaginar como os nossos arquivos de teste ficariam caso nossa aplicação fosse crescendo e novas funções fossem criadas? Se tivéssemos uma nova função adicionada, chamada `calculaDesconto`, no seguinte modelo:

```
const calculaDesconto = (salario, descontos) => {
    return salario - descontos;
};
```

Como será que ficaria a estrutura do nosso teste? Provavelmente teríamos que repetir tudo o que fizemos até aqui, duplicando nossa estrutura de `if/else` e nossas validações.

Um leve esboço disso seria algo mais ou menos assim:

```
const somaHorasExtras = (salario, valorHorasExtras) => {
    return salario + valorHorasExtras;
};
// nova função
const calculaDesconto = (salario, descontos) => {
    return salario - descontos;
};

// trocamos as variáveis para let, para podermos reatribuir
let esperado = 10;
let retorno = somaHorasExtras(5, 5);

if (esperado === retorno) {
    console.log(`? Teste passou`);
```

```
} else {
    console.error(`?? Ih, deu ruim...`);
}

// novo teste adicionado
esperado = 5;
retorno = calculaDesconto(10, 5);

if (esperado === retorno) {
    console.log(`? Teste passou`);
} else {
    console.error(`?? Ih, deu ruim...`);
}
```

Ao executarmos esse arquivo novamente, teremos o seguinte resultado:

```
? Teste passou
? Teste passou
```

Está funcionando perfeitamente. Mas podemos perceber alguns detalhes aqui:

- Conforme criamos novas funções, precisamos ficar duplicando nossa estrutura de testes;
- Caso algum dos testes falhe, não temos como saber qual é já que nossas mensagens de erro, atualmente, estão iguais.

Como temos um certo trabalho manual atualmente, vamos começar a criar algumas funções auxiliares para nos ajudar a testar nosso código.

Vamos criar um arquivo chamado `mini-framework.js` para esse próximo exemplo e nele vamos criar uma função chamada `teste`, que recebe três parâmetros: um `título`, um valor `esperado` e um valor a ser `comparado`.

```
const teste = (titulo, esperado, retornado) => {
    // vamos implementar a função teste
}
```

Vamos mover toda a nossa lógica de `if/else` para dentro dessa função também:

```
const teste = (titulo, esperado, retornado) => {
    if (esperado === retorno) {
        console.log(`? Teste passou`);
    } else {
        console.error(`?? Ih, deu ruim...`);
    }
}
```

Como vamos ter um título do teste, podemos utilizá-lo para customizarmos as mensagens de sucesso e erro também. Fica algo mais ou menos assim:

```
const teste = (titulo, esperado, retornado) => {
    if (esperado === retornado) {
        console.log(`?  ${titulo} passou`);
    } else {
        console.log(`?? ${titulo} deu ruim...`);
    }
}
```

Tudo o que precisamos fazer é aplicar essa função teste no lugar dos nossos `if/else`, que fizemos anteriormente.

Apenas para que tenhamos histórico e os exemplos prontos, vamos copiar as funções `somaHorasExtras` e `calculaDesconto` para o arquivo `mini-framework.js`, que acabamos de criar. Mais à frente reutilizaremos nossos códigos, mas, por ora, podemos cometer esse pequeno vacilo tecnológico:

```
const teste = (titulo, esperado, retornado) => {
    if (esperado === retornado) {
        console.log(`?  ${titulo} passou`);
    } else {
        console.log(`?? ${titulo} deu ruim...`);
    }
}
```

```
// copiamos as funções
const somaHorasExtras = (salario, valorHorasExtras) => {
    return salario + valorHorasExtras;
};

const calculaDesconto = (salario, descontos) => {
    return salario - descontos;
};
```

E vamos testá-las, apenas chamando a função `teste` com os valores que esperamos:

```
const teste = (titulo, esperado, retornado) => {
    if (esperado === retornado) {
        console.log(`?  ${titulo} passou`);
    } else {
        console.log(`?? ${titulo} deu ruim...`);
    }
}

const somaHorasExtras = (salario, valorHorasExtras) => {
    return salario + valorHorasExtras;
};

const calculaDesconto = (salario, descontos) => {
    return salario - descontos;
};

// adicionamos os dois testes
teste('somaHorasExtras', 10, somaHorasExtras(5, 5));
teste('calculaDesconto', 5, calculaDesconto(10, 5));
```

Vamos executar esse arquivo e, com isso, devemos ter a seguinte saída:

```
?  somaHorasExtras passou
?  calculaDesconto passou
```

Assim podemos criar nossos testes de maneira bem mais fácil e reutilizável!

3.3 Conhecendo asserções ao criar novos utilitários

Quando falamos sobre testes, existe um nome para indicar essa estrutura que verifica se um valor ou alguma entrada é igual a algum resultado esperado: isso se chama *asser o*.

Podemos trabalhar com asserções de diversas maneiras possíveis. Por enquanto, com o que fizemos até agora, conseguimos apenas comparar se um valor é exatamente igual a outro.

Vamos criar uma nova função para lidar especificamente com essas validações. Crie um novo arquivo chamado `mini-framework-com-assercoes.js` e vamos criar uma função chamada `verifiqueQue` , que recebe como parâmetro apenas um `valor` qualquer:

```
const verifiqueQue = (valor) => {
    // vamos implementar a função
}
```

O que essa função fará é nos retornar um objeto para montarmos nossas asserções contendo várias funções que utilizaremos como combinadores (ou *matchers*, em inglês). Em outras palavras, estamos montando uma estrutura para que possamos fazer uma asserção da seguinte forma:

```
verificaQue(algumValor).ehExatamenteIgualA(outroValor);
```

Para criarmos essa estrutura, precisamos que a função `verificaQue` retorne um objeto:

```
const verifiqueQue = (valor) => {
    const assercoes = {}; // criamos um objeto vazio

    return assercoes; // retornamos esse objeto
}
```

Para que esse objeto tenha as funções que vamos utilizar em nossas asserções, precisamos incluí-las nele.

Vamos criar uma função chamada `ehExatamenteIgualA`, para prosseguir:

```
const verifiqueQue = (valor) => {
    const assercoes = {
        ehExatamenteIgualA() { // criamos a função

        }
    };

    return assercoes;
}
```

Tudo o que precisamos fazer dentro dessa função é receber o valor `esperado` como parâmetro e fazer nossa validação com `if/else` como fizemos anteriormente:

```
const verifiqueQue = (valor) => {
    const assercoes = {
        ehExatamenteIgualA(esperado) {
            if (valor === esperado) {
                console.log(`?  ${titulo} passou`);
            } else {
                console.log(`?? ${titulo} deu ruim...`);
            }
        }
    };

    return assercoes;
}
```

Se tentarmos executar esse código, teremos um problema pois, atualmente, nossos `console.log` estão utilizando a variável `titulo`.

Vamos copiar nossa função `teste` para esse arquivo e refatorar um pouco, para mudarmos sua implementação.

```
const verifiqueQue = (valor) => {
    const assercoes = {
        ehExatamenteIgualA(esperado) {
            if (valor === esperado) {
```

```
                console.log(`?  ${titulo} passou`);
            } else {
                console.log(`?? ${titulo} deu ruim...`);
            }
        }
    };

    return assercoes;
}

// trouxemos a função teste do jeito que estava
const teste = (titulo, esperado, retornado) => {
    if (esperado === retornado) {
        console.log(`?  ${titulo} passou`);
    } else {
        console.log(`?? ${titulo} deu ruim...`);
    }
}
```

Vamos refatorar a função `teste`. Vamos apagar os parâmetros `esperado` e `retornado` e receber apenas `titulo` e um novo parâmetro chamado `funcaoTeste`:

```
const teste = (titulo, funcaoTeste) => { // ajustamos parâmetros
    if (esperado === retornado) {
        console.log(`?  ${titulo} passou`);
    } else {
        console.log(`?? ${titulo} deu ruim...`);
    }
}
```

Vamos imaginar que nosso teste será esse parâmetro novo que criamos (`funcaoTeste`). Em outras palavras, nosso teste será um `callback`, que será executado pela função `teste`. Estamos montando nossa *API* (interface/contrato das funções pela qual nossos testes devem funcionar) para funcionar dessa maneira:

```
teste('somaHorasExtras', () => {
    // nosso teste agora será executado aqui
});
```

Para alcançar esse resultado, tudo o que precisamos fazer é executar a função `funcaoTeste`. Vamos apagar nosso bloco de `if/else` e executar essa função:

```
const teste = (titulo, funcaoTeste) => {
    funcaoTeste();
}
```

Vamos englobar nossa `funcaoTeste` em um bloco de `try/catch`, pois logo depois ajustaremos também nossa função `verifiqueQue` para que ela dispare um erro caso nossas validações estejam erradas:

```
const teste = (titulo, funcaoTeste) => {
    try {
        funcaoTeste();
    } catch (err) {
    }
}
```

E vamos colocar novamente aquelas mensagens de sucesso/erro para deixar a visualização mais clara e mantermos nosso resultado impresso ao executar nos testes:

```
const teste = (titulo, funcaoTeste) => {
    try {
        funcaoTeste();
        console.log(`?  ${titulo} passou`);
    } catch (err) {
    console.log(`?? ${titulo} deu ruim...`);
    }
}
```

Para que nosso teste funcione no modelo que estamos construindo, vamos modificar nossa função `verificaQue`, para que ela dispare um erro caso os valores não sejam iguais:

```
const verifiqueQue = (valor) => {
    const assercoes = {
        ehExatamenteIgualA(esperado) {
            if (valor === esperado) {
                // não faremos nada caso os valores sejam iguais
```

```
            } else {
                throw {}; // dispararemos um erro vazio, apenas para ver o
funcionamento
            }
        }
    };

    return assercoes;
}

const teste = (titulo, funcaoTeste) => {
    try {
        funcaoTeste();
        console.log(`?  ${titulo} passou`);
    } catch (err) {
    console.log(`?? ${titulo} deu ruim...`);
    }
}
```

Podemos, inclusive, modificar nossa lógica dentro do `if/else`, já que temos um bloco `if` que não serve para mais nada. Vamos mudar nossa função para que o erro seja disparado diretamente se o valor for **diferente** do esperado:

```
const verifiqueQue = (valor) => {
    const assercoes = {
        ehExatamenteIgualA(esperado) {
            if (valor !== esperado) {
                 throw {};
            }
        }
    };

    return assercoes;
}
```

Bem melhor, não acha?

Aplicando esses novos utilitários

Com isso, podemos também modificar nossos testes para terem uma legibilidade um pouco melhor. Sabemos que agora precisaremos englobar tudo em um `callback` utilizando a função `teste`, então o código que anteriormente estava assim:

```
teste('somaHorasExtras', 10, somaHorasExtras(5, 5));
teste('calculaDesconto', 5, calculaDesconto(10, 5));
```

Ficará assim:

```
teste('somaHorasExtras', () => {
    // vamos implementar o teste
});
teste('calculaDesconto', () => {
    // vamos implementar o teste
});
```

Para que nossos testes funcionem, vamos trazer as funções `somaHorasExtras` e `calculaDesconto` para o arquivo `mini-framework-com-assercoes.js`:

```
const verifiqueQue = (valor) => {
    const assercoes = {
        ehExatamenteIgualA(esperado) {
            if (valor === esperado) {
            } else {
                throw {};
            }
        }
    };

    return assercoes;
}

const teste = (titulo, funcaoTeste) => {
    try {
        funcaoTeste();
        console.log(`?  ${titulo} passou`);
    } catch (err) {
    console.log(`?? ${titulo} deu ruim...`);
    }
```

```
}

// trouxemos as funções
const somaHorasExtras = (salario, valorHorasExtras) => {
    return salario + valorHorasExtras;
};

const calculaDesconto = (salario, descontos) => {
    return salario - descontos;
};

teste('somaHorasExtras', () => {
});
teste('calculaDesconto', () => {
});
```

Vamos refazer nossas asserções utilizando a função `verificaQue` e seu combinador `ehExatamenteIgualA` para validar se nossos valores estão corretos. Vamos criar nossas variáveis de valor `esperado` e `retornado` dentro dos nossos testes:

```
teste('somaHorasExtras', () => {
    const esperado = 10;
    const retornado = somaHorasExtras(5, 5);

});
teste('calculaDesconto', () => {
    const esperado = 5;
    const retornado = calculaDesconto(10, 5);
});
```

E vamos criar nossa asserção:

```
teste('somaHorasExtras', () => {
    const esperado = 10;
    const retornado = somaHorasExtras(5, 5);

    verifiqueQue(retornado).ehExatamenteIgualA(esperado);
});
teste('calculaDesconto', () => {
    const esperado = 5;
```

```
    const retornado = calculaDesconto(10, 5);

    verifiqueQue(retornado).ehExatamenteIgualA(esperado);
});
```

Com isso, nosso arquivo deve estar mais ou menos assim:

```
// funções de teste
const verifiqueQue = (valor) => {
    const assercoes = {
        ehExatamenteIgualA(esperado) {
            if (valor !== esperado) {
                throw {};
            }
        }
    };

    return assercoes;
}

const teste = (titulo, funcaoTeste) => {
    try {
        funcaoTeste();
        console.log(`? ${titulo} passou`);
    } catch (err) {
        console.log(`?? ${titulo} deu ruim...`);
    }
}

// funções a serem testadas
const somaHorasExtras = (salario, valorHorasExtras) => {
    return salario + valorHorasExtras;
};

const calculaDesconto = (salario, descontos) => {
    return salario - descontos;
};

// testes
teste('somaHorasExtras', () => {
    const esperado = 10;
```

```
    const retornado = somaHorasExtras(5, 5);

    verifiqueQue(retornado).ehExatamenteIgualA(esperado);
});
teste('calculaDesconto', () => {
    const esperado = 5;
    const retornado = calculaDesconto(10, 5);

    verifiqueQue(retornado).ehExatamenteIgualA(esperado);
});
```

Ao executar esse arquivo, teremos o seguinte resultado:

```
? somaHorasExtras passou
? calculaDesconto passou
```

Você pode estar se perguntando: "Mas para que fizemos toda essa abstração? Demos tantas voltas para chegar ao mesmo resultado?".

Justamente para que o entendimento de **asserções** fique um pouco mais claro. Podemos pensar em asserções como as validações que garantirão que nosso teste está passando como deveria e, por isso, teremos garantia de um teste **assertivo**.

Fora isso, também simulamos mais ou menos o funcionando do framework de testes que vamos utilizar daqui para a frente. Isso facilita o entendimento de algumas das "mágicas" que acontecem ao aplicar o Jest e como suas abstrações funcionam.

3.4 Trabalhando com assert nativo no NodeJS

O NodeJS também possui um módulo só para lidar com asserções chamado *assert* (https://nodejs.org/api/assert.html). Existe também a função `console.assert`, que funciona como um pequeno utilitário tanto no Node quanto no Navegador.

Vamos fazer um pequeno teste para entender como esse módulo funciona. Crie um arquivo novo chamado `assert-nativo.js` e importe este módulo:

```
// arquivo assert-nativo.js

const assert = require('assert');
```

Já podemos utilizá-lo para fazer asserções. Caso uma asserção seja válida, nada acontecerá. Caso ela esteja incorreta, receberemos um erro.

Dentre os métodos que podemos utilizar para realizar o que queremos, vamos escolher o `strictEqual`. Com isso, vamos fazer um teste com uma asserção válida (comparando 5 com 5) e outro com uma inválida (comparando 5 com 10):

```
// arquivo assert-nativo.js

const assert = require('assert');

assert.strictEqual(5, 5); // asserção válida
assert.strictEqual(5, 10); // asserção inválida
```

Execute esse arquivo com o comando `node assert-nativo.js` e veja a seguinte saída no terminal:

```
throw new AssertionError(obj);
  ^

AssertionError [ERR_ASSERTION]: 5 == 10
    at Object.<anonymous>
(/Users/gabriel.ramos/Development/personal/javascriptassertivo.com.br/proj
etos/01-fundamentos-de-testes/02-frameworks-e-ambientes/04-assert-
nativo.js:4:8)
    at Module._compile (internal/modules/cjs/loader.js:1076:30)
    at Object.Module._extensions..js
(internal/modules/cjs/loader.js:1097:10)
    at Module.load (internal/modules/cjs/loader.js:941:32)
    at Function.Module._load (internal/modules/cjs/loader.js:782:14)
```

```
    at Function.executeUserEntryPoint [as runMain]
(internal/modules/run_main.js:72:12)
    at internal/main/run_main_module.js:17:47 {
  generatedMessage: true,
  code: 'ERR_ASSERTION',
  actual: 5,
  expected: 10,
  operator: '=='
}
-----
```

Perfeito! Recebemos o erro indicando que nossa segunda asserção (que compara 5 com 10) é inválida. Bem bacana, não?

Caso queira se aprofundar um pouco mais nesse módulo nativo, proponho um desafio para você: modifique o arquivo `mini-framework-com-assercoes.js` para utilizar esse `assert` nativo com essa mesma função `strictEqual`, em vez de ficar disparando erros manualmente na função `ehExatamenteIgualA`.

Aproveite para mostrar a mensagem de erro junto ao `console.log`, que é executado caso o teste falhe também!

Com tudo isso, temos nossa versão de um miniframework e agora podemos conhecer o Jest, que nos ajudará na nossa missão de realizar nossos testes.

Tire um tempo para brincar com esse exemplo, modificar as funções e ver os testes passando ou falhando. Acabamos de aprender conceitos bem importantes que servirão de base para tudo o que veremos a seguir.

Capítulo 4
Diga olá ao Jest!

Ao longo dos próximos módulos, para nossos testes de unidade e integração, utilizaremos uma ferramenta chamada Jest (https://jestjs.io/).

Jest é um framework de testes construído pelo Facebook que alcançou bastante visibilidade no mercado nos últimos anos por sua simplicidade. Por ser uma ferramenta muito completa, ele nos traz uma série de facilidades e praticamente tudo de que precisamos em um ambiente de testes.

Veremos a seguir como é fácil criar um teste com Jest e como a estrutura do miniframework que criamos se assemelha à forma como ele funciona. O código deste capítulo será o da pasta `projetos/01-fundamentos-de-testes/03-ola-jest` no repositório em que estamos trabalhando: https://javascriptassertivo.com.br/.

4.1 Instalação e primeiros passos

Vamos criar um novo projeto para que possamos configurar o Jest e entender como ele funciona.

Crie uma nova pasta e inicie um novo projeto com aquele comando que vimos anteriormente:

```
npm init -y
```

Com a estrutura inicial pronta e o arquivo `package.json` gerado, vamos instalar o Jest. Assim como o ESLint, o Jest é um pacote que só utilizaremos enquanto desenvolvemos a nossa aplicação, ou seja, não é uma dependência da qual nossa aplicação ou nossos projetos precisariam para funcionar em ambientes de produção. Por

isso, vamos instalá-lo como `devDependencies`, com o seguinte comando:

```
npm install --save-dev jest
# ou, de maneira mais curta
npm i -D jest
```

Se tentarmos rodar o comando de teste:

```
npm test
# ou
npm t
```

Teremos um erro:

```
Error: no test specified
npm ERR! Test failed.  See above for more details
```

Isso acontece porque ainda não configuramos o que queremos rodar no comando de `test` nos `scripts` do nosso projeto. Para utilizar o Jest com esse comando, vamos fazer um pequeno ajuste no arquivo `package.json` arrumando exatamente esse `script`:

```
"scripts": {
    "test": "jest" // editamos esse script para executar o Jest
},
```

Agora sim, podemos executar o comando `test`:

```
npm test
```

E podemos observar que teremos outra mensagem como saída no nosso terminal:

```
No tests found, exiting with code 1
Run with `--passWithNoTests` to exit with code 0
In
/Users/gabriel.ramos/Development/personal/javascriptassertivo.com.br/proje
tos/01-fundamentos-de-testes/03-ola-jest
  2 files checked.
  testMatch: **/__tests__/**/*.[jt]s?(x), **/?(*.)+(spec|test).[tj]s?(x) -
0 matches
```

```
  testPathIgnorePatterns: /node_modules/ - 2 matches
  testRegex:  - 0 matches
Pattern:  - 0 matches
npm ERR! Test failed.  See above for more details.
```

O que aconteceu é que executamos o Jest com sucesso, mas não conseguimos testar nada, pois nosso projeto está zerado. Como acabamos de criar uma nova pasta e instalar o Jest, ainda não existem arquivos a serem testados.

Porém, antes de criarmos qualquer arquivo, podemos dar uma olhada com mais atenção nessa mensagem de erro pois ela já indica algumas formas bem intuitivas de trabalhar com o Jest. Vamos olhar atentamente para a linha que indica o `testMatch` :

```
testMatch: **/__tests__/**/*.[jt]s?(x), **/?(*.)+(spec|test).[tj]s?(x) - 0
matches
```

O que essa linha nos informa é que o Jest tentou encontrar alguns arquivos seguindo alguns padrões de nome/pasta. Por isso, temos essas duas expressões regulares e, no fim, um indicativo de que 0 (nenhum) arquivo combina com esse padrão.

Em resumo, o que essas expressões regulares indicam é que o Jest já verifica, por padrão, qualquer arquivo de teste finalizado com extensão `.js` , `.jsx` , `.ts` ou . `tsx` , sejam eles localizados ou não em uma pasta `__tests__` ou com um sufixo `.spec` ou `.test` .

Isso já é uma certa convenção de mercado. É comum que testes estejam em um diretório ou subdiretório com `__tests__` ou tenham nomenclaturas como `meu-arquivo.test.js` ou `meu-arquivo.spec.js` . As extensões podem variar dependendo do seu projeto. É comum encontrar por aí arquivos com extensão `.jsx` (componentes React) e também `.tsx` (componentes React com TypeScript).

4.2 Configurações iniciais e conhecendo a CLI

Assim como o ESLint, o Jest possui uma *CLI* própria que nos ajuda em várias tarefas. Mas o que afinal é uma *CLI*?

CLI significa *command-line interface* (ou interface de linha de comando, em português) e é uma forma de nomear esses programas que executamos via terminal. Isso quer dizer que você já estava manipulando algumas CLIs e não sabia disso!

Por si só esse já é um assunto à parte bem interessante e, caso você tenha curiosidade, aprender a criar uma CLI própria pode te ajudar a entender conceitos importantes. É um ótimo tópico de estudo e, inclusive, aplicaremos testes unitários em uma CLI no próximo capítulo.

Tudo o que precisaremos saber por enquanto é que poderemos executar o Jest (assim como inúmeras outras ferramentas) através do nosso terminal e também informar alguns argumentos extras. Inclusive a documentação da CLI do Jest https://jestjs.io/docs/pt-BR/cli é bem completa.

Para que possamos entender alguns desses argumentos, vamos criar novamente um arquivo contendo nossas funções `somaHorasExtras` e `calculaDescontos`. Podemos chamar esse arquivo de `operacoes.js`:

```
// arquivo operacoes.js
const somaHorasExtras = (salario, valorHorasExtras) => {
    return salario + valorHorasExtras;
};

const calculaDesconto = (salario, descontos) => {
    return salario - descontos;
};
```

Vamos exportar essas duas funções utilizando o padrão *CommonJS* ao final do arquivo:

```
// arquivo operacoes.js
```

```
const somaHorasExtras = (salario, valorHorasExtras) => {
    return salario + valorHorasExtras;
};

const calculaDesconto = (salario, descontos) => {
    return salario - descontos;
};

// exportamos nossas funções
module.exports = {
    somaHorasExtras,
    calculaDesconto
};
```

Vamos criar um arquivo de teste para testar essas duas funções utilizando o Jest. Vamos nomeá-lo `operacoes.test.js` e importar essas duas funções que exportamos anteriormente:

```
// arquivo operacoes.test.js

// importamos o módulo operações e extraímos as funções que criamos
const { calculaDesconto, somaHorasExtras } = require('./operacoes');
```

Se rodarmos nossos testes agora com `npm test`, teremos o seguinte erro:

```
> jest

 FAIL  ./operacoes.test.js
  ? Test suite failed to run

    Your test suite must contain at least one test.
```

Isso acontece porque ainda não escrevemos nenhum teste. Perfeito!

O Jest (e várias outras ferramentas de testes) se refere aos testes como *suítes*. Podemos pensar que suítes de testes, para o Jest, são basicamente os arquivos de teste que existem no projeto.

Podemos deixar o Jest assistindo aos nossos arquivos enquanto escrevemos esses testes, para nos poupar o trabalho de ficar

executando `npm test` a cada alteração que fazemos em um arquivo. Para fazer isso, basta passarmos o parâmetro `--watch` na CLI do Jest.

Como o script que executa o Jest é um script configurado dentro do nosso projeto, precisaremos passar esse argumento para a CLI após uma sequência de `--` extra ou, como vimos anteriormente, executando direto via NPX da seguinte forma:

```
npm test -- --watch
# ou, caso queira executar o jest diretamente
npx jest --watch
```

Ao executar esse comando, uma nova tela aparecerá. Ainda teremos o erro já que não criamos nenhum teste, mas teremos algumas novas opções para interagir com esse novo processo que assiste aos nossos arquivos:

```
Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

Essas opções nos serão muito úteis na nossa jornada com testes. Elas servem para que possamos guiar esse processo de assistência nos nossos testes. Ao traduzi-las, temos algo como:

- Aperte `a` para rodar todos os testes;
- Aperte `f` para rodar apenas os testes que falharam;
- Aperte `p` para filtrar por um nome de arquivo, seguindo alguma expressão regular;
- Aperte `t` para filtrar pelo nome de algum teste, seguindo alguma expressão regular;
- Aperte `q` para sair desse modo de assistência;
- Aperte `enter` para executar os testes.

Agora, a cada vez que salvarmos os arquivos de teste ou o arquivo que possui o código que estamos testando, o próprio Jest reexecutará nossos testes. Bem melhor, não acha?

Para facilitar um pouco nossa vida, podemos criar um script extra dentro do nosso `package.json` para realizar esse trabalho de executar o Jest no modo de assistência. Isso vai nos poupar alguns caracteres e a necessidade de ficar executando `--watch` o tempo todo. Podemos chamá-lo talvez de `test:watch`. Vamos colocá-lo no `package.json`:

```
"scripts": {
  "test": "jest",
    "test:watch": "jest --watch",
},
```

Para executar esse mesmo processo que ficará assistindo aos arquivos enquanto editamos, basta executar:

```
npm run test:watch
```

Bem mais prático, não?

Vamos escrever um teste para remover esse erro que estamos recebendo e começar a entender a estrutura do Jest. Execute o comando que criamos no terminal e vamos para o nosso primeiro teste.

4.3 Primeiro teste

Vamos criar nosso teste no arquivo `operacoes.test.js`. Para nos ajudar nesta etapa, podemos lembrar um pouco da estrutura que fizemos no capítulo anterior com nosso miniframework:

```
teste('somaHorasExtras', () => {
    const esperado = 10;
    const retornado = somaHorasExtras(5, 5);
```

```
    verifiqueQue(retornado).ehExatamenteIgualA(esperado);
});
```

A estrutura que usamos aqui é bem semelhante à do Jest. Para criarmos um teste, precisaremos utilizar uma função chamada `test`, que também recebe uma string, e uma função de *callback* como parâmetro. Vamos começar a criar essa função no nosso arquivo:

```
const { calculaDesconto, somaHorasExtras } = require('./operacoes');

test('somaHorasExtras', () => {
    // vamos escrever o teste
});
```

Precisaremos então trabalhar com as nossas asserções. O Jest nos entrega uma série de asserções com as quais podemos trabalhar utilizando a função `expect`. Essa função receberá um valor e nos retornará um objeto com várias outras funções que podemos combinar para montar nossas asserções. É bem parecido com o exemplo que criamos na linha:

```
verifiqueQue(retornado).ehExatamenteIgualA(esperado);
```

Podemos então criar nossas variáveis com os valores `esperado` e `retornado` e passar, por exemplo, o valor de `retornado` para a função `expect`:

```
const { calculaDesconto, somaHorasExtras } = require('./operacoes');

test('somaHorasExtras', () => {
    const esperado = 10;
    const retornado = somaHorasExtras(5, 5);

    expect(retornado) // vamos criar a combinação
});
```

Agora, precisamos apenas trabalhar com alguma combinação que é retornada pela função `expect` para que possamos validar que nosso

valor `retornado` é igual ao `esperado`. Para esse caso, podemos utilizar a função `toBe` da seguinte maneira:

```
const { calculaDesconto, somaHorasExtras } = require('./operacoes');

test('somaHorasExtras', () => {
    const esperado = 10;
    const retornado = somaHorasExtras(5, 5);

    expect(retornado).toBe(esperado);
});
```

Salve o arquivo editado e você perceberá que o teste anterior passou, e agora nosso teste funciona como deveria:

```
PASS  ./operacoes.test.js
  ? somaHorasExtras (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.411 s, estimated 1 s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

Como temos somente um arquivo e, portanto, somente uma suíte de teste, teremos como resultado que essa suíte passou nos testes escritos.

Melhorando a legibilidade e a escrita dos testes

Se pararmos para ler atentamente o que acabamos de escrever, perceberemos que nosso teste tem uma certa didática em sua leitura. Se fôssemos traduzir para português a estrutura que acabamos de criar, poderíamos ler algo mais ou menos assim: **teste de `somaHorasExtras`, espero que o valor retornado `somaHorasExtras(5, 5)` seja igual ao esperado `(10)`.**

Assim fica bem claro como nossa função deve se comportar, não acha? Por isso, muitas pessoas acabam dizendo que um teste bem estruturado também serve como uma pequena estrutura de documentação, já que ajuda a indicar o funcionamento de um sistema.

Podemos, inclusive, deixar o título do teste um pouco mais claro. Que tal trocá-lo para algo como:

```
const { calculaDesconto, somaHorasExtras } = require('./operacoes');

test('deve somar horas extras', () => {
    const esperado = 10;
    const retornado = somaHorasExtras(5, 5);

    expect(retornado).toBe(esperado);
});
```

Fica ainda mais descritivo, não? Para nos ajudar com essa questão de leitura, temos uma função chamada `it`. Ela é apenas um apelido para a função `test`, que já aplicamos, então funciona da mesma maneira. Ela apenas muda a forma como nosso teste é lido. Vamos realizar essa mudança:

```
const { calculaDesconto, somaHorasExtras } = require('./operacoes');

// trocamos "test" por "it"
it('deve somar horas extras', () => {
    const esperado = 10;
    const retornado = somaHorasExtras(5, 5);

    expect(retornado).toBe(esperado);
});
```

Não ficou bem melhor? Agora nossa leitura, fica algo como: **isso deve somar horas extras e espera-se que o valor retornado `somaHorasExtras(5, 5)` seja igual ao esperado `(10)`**. Em que "isso" (ou *it*) é usado para fazer referência à própria função que está sendo testada.

Quando os testes são escritos em inglês, a leitura faz um pouco mais de sentido, já que ficaria assim: **it should sum extra hours**. Mas mesmo em português acho que é uma estrutura completamente válida de se manter.

É muito importante manter uma boa legibilidade nos testes que escrevemos. Isso nos ajuda enquanto estamos desenvolvendo, ajuda as novas pessoas que entrarem no time a entender o código que já existe, facilita processos de depuração de erros e também ajuda a encontrar testes quebrados mais facilmente.

Portanto, vamos tentar manter a legibilidade mais simples e prática possível em nossos testes, ok?

Vamos criar um teste para a função `calculaDesconto`, seguindo a mesma estrutura:

```
const { calculaDesconto, somaHorasExtras } = require('./operacoes');

it('deve somar horas extras', () => {
    const esperado = 10;
    const retornado = somaHorasExtras(5, 5);

    expect(retornado).toBe(esperado);
});

// novo teste
it('deve calcular descontos', () => {
    const esperado = 5;
    const retornado = calculaDesconto(10, 5);

    expect(retornado).toBe(esperado);
});
```

Ao salvar o arquivo, perceberemos que uma nova linha aparecerá contendo o teste que acabamos de criar:

```
PASS  ./operacoes.test.js
  ? deve somar horas extras (1 ms)
  ? deve calcular descontos (1 ms)
```

```
Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.548 s, estimated 1 s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

Como esses dois testes estão relacionados a um mesmo contexto, podemos, opcionalmente, agrupá-los com uma função chamada `describe`. Essa função é apenas um utilitário para encapsularmos esses testes dentro de um determinado escopo. Vamos aplicá-la englobando os nossos dois testes.

Sua estrutura é exatamente igual à da função `it` ou `test`, recebendo como parâmetro um título e uma função de *callback*, onde deveremos colocar os dois testes que criamos, da seguinte forma:

```
const { calculaDesconto, somaHorasExtras } = require('./operacoes');

// criamos o bloco com describe e inserimos um título
describe('Operações', () => {
    // movemos o primeiro teste para dentro
    it('deve somar horas extras', () => {
    const esperado = 10;
    const retornado = somaHorasExtras(5, 5);

    expect(retornado).toBe(esperado);
    });

    // movemos o segundo teste para dentro
    it('deve calcular descontos', () => {
    const esperado = 5;
    const retornado = calculaDesconto(10, 5);

    expect(retornado).toBe(esperado);
    });
});
```

Ao salvar esse arquivo, perceberemos que nossa mensagem agora conterá o título `Operações` agrupando nossos testes:

```
Operações
    ? deve somar horas extras (2 ms)
    ? deve calcular descontos

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.372 s
Ran all test suites related to changed files.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

Toda essa estrutura de `describe` pode ser criada da forma como você quiser. Você pode colocar funções `describe` dentro de outras funções `describe` sem problema nenhum, sempre que fizer sentido e for ajudar a estrutura do seu teste.

4.4 Executando tarefas repetitivas com Hooks

O Jest possui algumas funções chamadas *Hooks* (ou ganchos, em português), que são utilizadas para executar um trecho qualquer de código em certas etapas no "ciclo de vida" de um teste. Um ciclo de vida é, basicamente, um determinado momento antes ou depois da execução de seu teste.

Esses *hooks* calham muito bem quando precisamos limpar algum dado que criamos ao iniciar um teste ou executar alguma configuração repetitiva.

Existem quatro *hooks* para utilização:

- `beforeAll` : para executar algo antes da execução de todos os testes.
- `beforeEach` : para executar algo antes de cada um dos testes iniciar.
- `afterEach` : para executar algo após a finalização de cada um dos testes.
- `afterAll` : para executar algo após finalizar todos os testes.

Cada uma dessas funções deve ser executada recebendo uma outra função como *callback*.

Para que possamos entender como eles, de fato, funcionam, vamos adicionar os quatro ao teste que criamos apenas para exibir um log no nosso terminal:

```
const { calculaDesconto, somaHorasExtras } = require('./operacoes');

describe('Operações', () => {
  beforeAll(() => {}); // para executar antes de todos os testes
  afterAll(() => {}); // para executar após todos os testes
  beforeEach(() => {}); // para executar antes de cada um dos testes
iniciar
  afterEach(() => {}); // para executar após cada um dos testes finalizar

  it('deve somar horas extras', () => {
    const esperado = 10;
    const retornado = somaHorasExtras(5, 5);

    expect(retornado).toBe(esperado);
  });

  it('deve calcular descontos', () => {
    const esperado = 5;
    const retornado = calculaDesconto(10, 5);

    expect(retornado).toBe(esperado);
  });
});
```

Vamos colocar um `console.log` dentro de cada uma das funções, indicando a etapa onde elas devem ser executadas, para vermos o seu funcionamento:

```
const { calculaDesconto, somaHorasExtras } = require('./operacoes');

describe('Operações', () => {
  // console.log com mensagem dentro de cada um dos hooks
  beforeAll(() => {
    console.log('Hook antes de todos os testes');
  });
  afterAll(() => {
    console.log('Hook após todos os testes');
  });
  beforeEach(() => {
    console.log('Hook antes de cada um dos testes iniciar');
  });
  afterEach(() => {
    console.log('Hook após cada um dos testes finalizar');
  });

  it('deve somar horas extras', () => {
    const esperado = 10;
    const retornado = somaHorasExtras(5, 5);

    expect(retornado).toBe(esperado);
  });

  it('deve calcular descontos', () => {
    const esperado = 5;
    const retornado = calculaDesconto(10, 5);

    expect(retornado).toBe(esperado);
  });
});
```

A saída do nosso terminal deve ter ficado um pouco poluída, mas, se olharmos os *logs*, teremos:

```
console.log
    Hook antes de todos os testes
```

```
    at Object.<anonymous> (operacoes.test.js:6:11)

console.log
  Hook antes de cada um dos testes iniciar

    at Object.<anonymous> (operacoes.test.js:15:12)

console.log
  Hook após cada um dos testes finalizar

    at Object.<anonymous> (operacoes.test.js:18:13)

console.log
  Hook antes de cada um dos testes iniciar

    at Object.<anonymous> (operacoes.test.js:15:12)

console.log
  Hook após cada um dos testes finalizar

    at Object.<anonymous> (operacoes.test.js:18:13)

console.log
  Hook após todos os testes

    at Object.<anonymous> (operacoes.test.js:11:6)
```

Com isso, podemos identificar que:

- A mensagem `Hook antes de todos os testes` foi exibida somente no início.
- A mensagem `Hook antes de cada um dos testes iniciar` apareceu duas vezes, uma para cada início dos testes.
- A mensagem `Hook após cada um dos testes finalizar` apareceu duas vezes, uma ao final de cada um dos testes.
- A mensagem `Hooks após todos os testes` apareceu uma vez, após a finalização de todos os testes.

Exatamente como esperávamos!

Esses *hooks* ainda vão nos ajudar bastante na nossa jornada. Por enquanto, temos apenas esse exemplo para que possamos entender como eles funcionam.

4.5 Como ler o relatório de testes

É comum que ferramentas de testes disponibilizem um relatório (ou *report*) contendo informações sobre os testes que foram executados. Entender esses relatórios e saber como utilizá-los da melhor forma possível nos ajuda muito na tarefa de criar confiança no código que escrevemos.

Para exibir esse relatório utilizando o Jest, podemos utilizar o argumento `--coverage` ao executar os testes.

Vamos criar um novo script para isso no nosso `package.json` :

```
"scripts": {
    "test": "jest",
    "test:watch": "jest --watch",
    "test:coverage": "jest --coverage" // criamos novo script
},
```

E então, basta executar `npm run test:coverage` , que teremos o seguinte resultado no nosso terminal:

```
--------------|---------|----------|---------|---------|------------------
-
File          | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
--------------|---------|----------|---------|---------|------------------
-
All files     |     100 |      100 |     100 |     100 |
 operacoes.js |     100 |      100 |     100 |     100 |
--------------|---------|----------|---------|---------|------------------
-
Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
```

```
Snapshots:   0 total
Time:        1.486 s
Ran all test suites.
```

Percebemos que agora temos uma tabela contendo um relatório dos nossos testes com algumas colunas e linhas:

- `File` , ou arquivo: os arquivos de teste que foram executados nos testes.
- `Stmts` , ou declarações: indica a porcentagem dos termos de declaração (como variáveis e imports) que foram ou não cobertos ao executar os testes.
- `Branch` , ou ramificações: indica a porcentagem de ramificações (como blocos if/else ou switch/case) que foram executadas ao longo dos testes.
- `Funcs` , ou funções: indica a porcentagem de funções que foram executadas ao longo dos testes;
- `Lines` , ou linhas: indica a porcentagem de linhas que foram executadas ao longo dos testes;
- `Uncovered Lines` , ou linhas descobertas: indica quais linhas de determinado arquivo não foram executadas pelos testes.

Além disso, podemos ver esse relatório de uma forma mais interativa. Não sei se você reparou, mas, após executar esse último comando, uma pasta `coverage` foi criada junto aos arquivos que testamos.

Ao abrir a pasta `coverage` e acessar sua subpasta `lcov-report` , vemos um arquivo `index.html` . Abrindo-o, é possível visualizar esse mesmo relatório e navegar pelos arquivos testados:

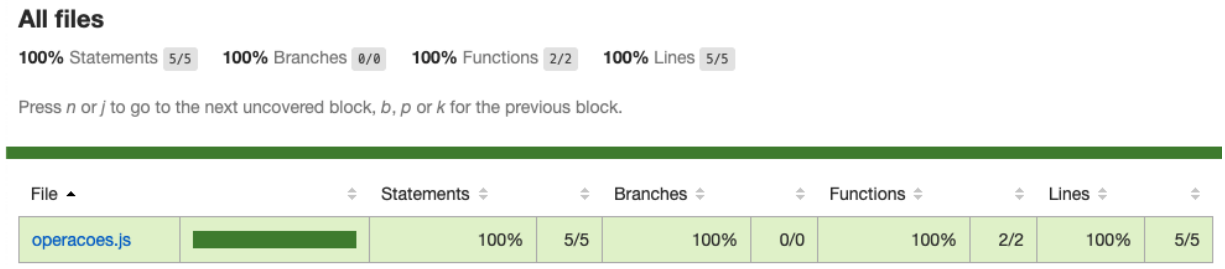


Figura 4.1: Relatório de testes.

Clicando, por exemplo, no link que está no nome do arquivo `operacoes.js` , podemos ver o arquivo testado:

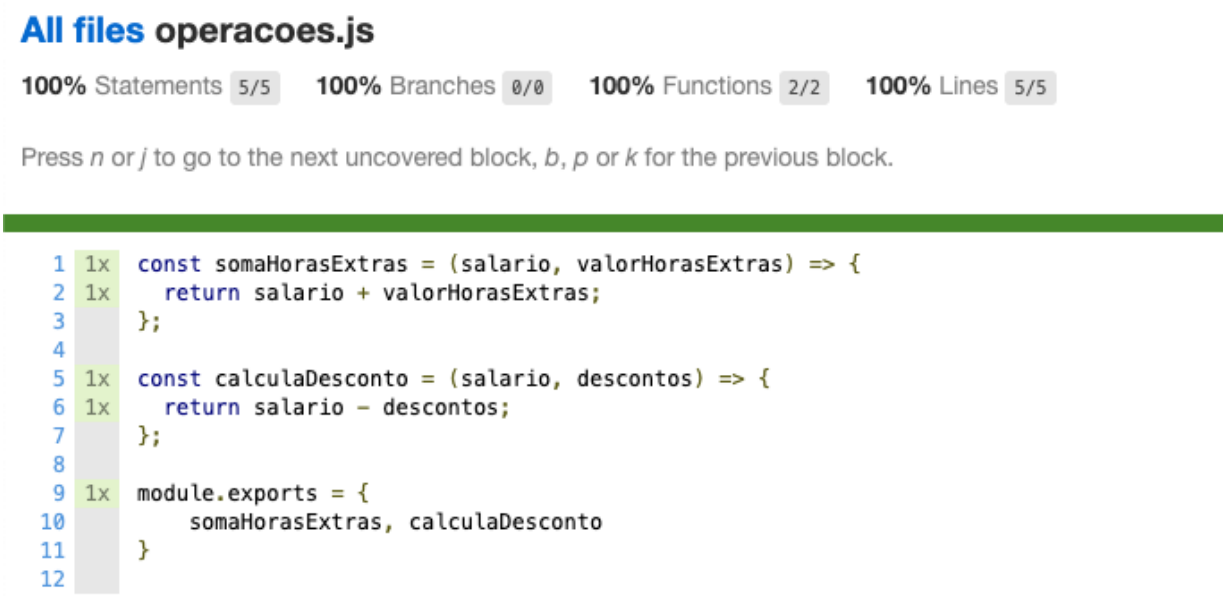

All files operacoes.js
100% Statements 5/5
100% Branches 0/0
100% Functions 2/2
100% Lines 5/5
Press n or j to go to the next uncovered block, b, p or k for the previous block.
1 1x const somaHorasExtras = (salario, valorHorasExtras) => {
2 1x return salario + valorHorasExtras;
3 };
4
5 1x const calculaDesconto = (salario, descontos) => {
6 1x return salario - descontos;
7 };
8
9 1x module.exports = {
10 somaHorasExtras, calculaDesconto
11 }
12

Figura 4.2: Relatório de testes - Arquivo testado.

Vamos modificar nosso teste no arquivo `operacoes.test.js` para ver como esse relatório se comportará.

Podemos indicar que o Jest deve pular algum teste diretamente na função `it` . Vamos utilizar a função `it.skip` no teste da função de descontos, por exemplo:

```
const { calculaDesconto, somaHorasExtras } = require('./operacoes');

describe('Operações', () => {
```

```
  beforeAll(() => {
    console.log('Hook antes de todos os testes');
  });
  afterAll(() => {
    console.log('Hook após todos os testes');
  });
  beforeEach(() => {
    console.log('Hook antes de cada um dos testes iniciar');
  });
  afterEach(() => {
    console.log('Hook após cada um dos testes finalizar');
  });

  it('deve somar horas extras', () => {
    const esperado = 10;
    const retornado = somaHorasExtras(5, 5);

    expect(retornado).toBe(esperado);
  });

    // adicionamos o .skip para pular o teste
  it.skip('deve calcular descontos', () => {
    const esperado = 5;
    const retornado = calculaDesconto(10, 5);

    expect(retornado).toBe(esperado);
  });
});
```

Se preferirmos, podemos também fazer o inverso informando ao Jest um único teste para executar nesse arquivo, com a função `it.only`:

```
const { calculaDesconto, somaHorasExtras } = require('./operacoes');

describe('Operações', () => {
  // console.log com mensagem dentro de cada um dos hooks
  beforeAll(() => {
    console.log('Hook antes de todos os testes');
  });
  afterAll(() => {
```

```
    console.log('Hook após todos os testes');
  });
  beforeEach(() => {
    console.log('Hook antes de cada um dos testes iniciar');
  });
  afterEach(() => {
    console.log('Hook após cada um dos testes finalizar');
  });

    // adicionamos o it.only
  it.only('deve somar horas extras', () => {
    const esperado = 10;
    const retornado = somaHorasExtras(5, 5);

    expect(retornado).toBe(esperado);
  });

    // removemos o it.skip
  it('deve calcular descontos', () => {
    const esperado = 5;
    const retornado = calculaDesconto(10, 5);

    expect(retornado).toBe(esperado);
  });
});
```

Ambos os resultados seriam iguais, já que temos apenas dois testes nos arquivos. Mas quando nossos testes começam a crescer e queremos executar apenas um teste em um arquivo ou pular alguns específicos, essas funções podem nos ajudar.

Escolha a que for mais conveniente para você e rode o comando para gerar o relatório de cobertura novamente:

```
npm run test:coverage
```

Vamos olhar nossos dois relatórios novamente. Primeiro, o do terminal:

```
-------------|---------|----------|---------|---------|------------------
-
```

```
File          | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
--------------|---------|----------|---------|---------|------------------
-
All files     |      80 |      100 |      50 |      80 |
 operacoes.js |      80 |      100 |      50 |      80 | 6
--------------|---------|----------|---------|---------|------------------
-
Test Suites: 1 passed, 1 total
Tests:       1 skipped, 1 passed, 2 total
Snapshots:   0 total
Time:        1.149 s
```

Podemos perceber que os valores das colunas `Stmts`, `Branch`, `Funcs` e `Lines` reduziram, justamente porque nosso teste não foi executado sobre a função de calcular descontos. Também podemos verificar que a linha `6` do nosso arquivo `operacoes.js` não foi executada, já que pulamos esse teste.

Se olharmos o relatório em HTML, teremos o mesmo resultado:

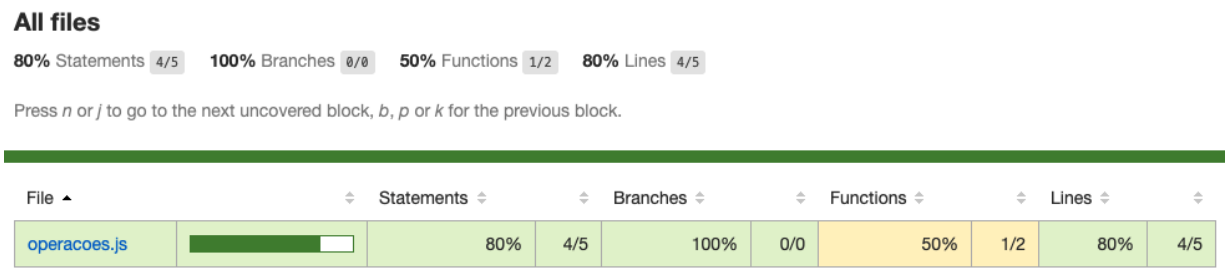


Figura 4.3: Relatório de testes - Parcialmente testado.

E após clicar no link `operacoes.js`:

All files operacoes.js
80% Statements 4/5
100% Branches 0/0
50% Functions 1/2
80% Lines 4/5
Press n or j to go to the next uncovered block, b, p or k for the previous block.
1 1x const somaHorasExtras = (salario, valorHorasExtras) => {
2 1x return salario + valorHorasExtras;
3 };
4
5 1x const calculaDesconto = (salario, descontos) => {
6 return salario - descontos;
7 };
8
9 1x module.exports = {
10 somaHorasExtras, calculaDesconto
11 }
12

Figura 4.4: Relatório de testes - Arquivo parcialmente testado

Bem útil, não acha?

Esses relatórios são muito importantes para nós, pessoas, e também para integrar possíveis serviços que analisam cobertura de teste.

4.6 Cobertura e a corrida pelo 100%

Aproveitando que tocamos no termo **cobertura**, vamos entendê-lo um pouco melhor.

Se você notou, o argumento que informamos ao Jest para gerar esse relatório (o `coverage`) significa justamente **cobertura** em inglês. Dizemos que um código tem sua cobertura ou está coberto por testes quando, justamente através de nossos testes, suas linhas foram executadas.

Entretanto, é comum olhar para um relatório de testes e acreditar que seu código está bem testado só porque ele indica que sua

cobertura é, por exemplo, de 100%. Isso é um completo engano.

É muito fácil gerar **falsos positivos** em testes e isso quer dizer que devemos nos atentar ao analisar esses relatórios para testar nosso código de forma coerente.

Vamos simular um cenário criando um arquivo chamado `falso-positivo.js` contendo as duas funções seguintes:

```
const funcaoInterna = () => {
  console.log('salvar algum dado');
}

const falsoPositivo = () => {
  funcaoInterna();
  return 'texto qualquer';
}

module.exports = {
  falsoPositivo
};
```

Exportamos somente a função `falsoPositivo`, e a `funcaoInterna` não pode ser acessada fora do módulo criado.

Vamos criar um arquivo de teste chamado `falso-positivo.test.js`, importar a função `falsoPositivo` e fazer um teste bem simples para ela, verificando se ela retorna uma string qualquer. Para fazer isso, podemos utilizar o método `.any` da própria função `expect`. Passamos para esse método um construtor de algum tipo de variável, que em nosso caso será `String`, da seguinte forma:

```
it('retorna um texto qualquer', () => {
  expect(falsoPositivo()).toEqual(expect.any(String));
});
```

Se rodarmos nosso comando para gerar o relatório de cobertura agora, veremos que tudo está 100% coberto, ou seja, todo o código que criamos foi executado:

```
-------------------|---------|----------|---------|---------|-------------
------
File               | % Stmts | % Branch | % Funcs | % Lines | Uncovered
Line #s
-------------------|---------|----------|---------|---------|-------------
------
All files          |     100 |      100 |     100 |     100 |
 falso-positivo.js |     100 |      100 |     100 |     100 |
 operacoes.js      |     100 |      100 |     100 |     100 |
-------------------|---------|----------|---------|---------|-------------
------

Test Suites: 2 passed, 2 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.22 s
Ran all test suites.
```

E no arquivo HTML gerado, podemos ver que o arquivo `falso-positivo.js` e seu conteúdo está totalmente bem:

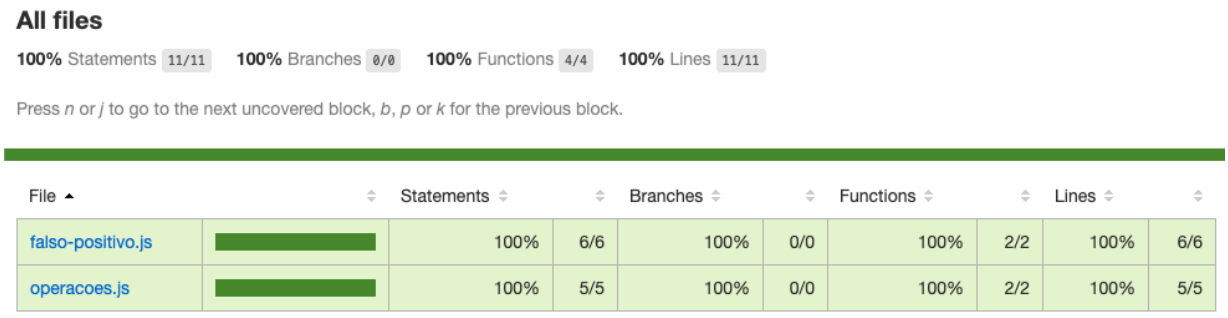

All files
100% Statements 11/11
100% Branches 0/0
100% Functions 4/4
100% Lines 11/11
Press n or j to go to the next uncovered block, b, p or k for the previous block.
File
Statements
Branches
Functions
Lines
falso-positivo.js 100% 6/6 100% 0/0 100% 2/2 100% 6/6
operacoes.js 100% 5/5 100% 0/0 100% 2/2 100% 5/5

Figura 4.5: Relatório de testes - Listagem de arquivos com falso positivo.

All files falso-positivo.js
100% Statements 6/6
100% Branches 0/0
100% Functions 2/2
100% Lines 6/6
Press n or j to go to the next uncovered block, b, p or k for the previous block.
1 1x const funcaoInterna = () => {
2 1x console.log('salvar algum dado');
3 }
4
5 1x const falsoPositivo = () => {
6 1x funcaoInterna();
7 1x return 'texto qualquer';
8 }
9
10 1x module.exports = {
11 falsoPositivo
12 };
13

Figura 4.6: Relatório de testes - Arquivo com falso positivo.

Entretanto, claramente não testamos nada sobre a `funcaoInterna`. Ela apenas é indicada como coberta porque nossa função `falsoPositivo` a executa internamente.

Ou seja, mesmo o código da `funcaoInterna` tendo sido executado, não possuímos nenhum teste para garantir que ela funciona como deveria e nem mesmo para garantir que a função `falsoPositivo` a está executando.

O que eu quero alertar com tudo isso é que é muito fácil se deixar enganar por relatórios que trazem cobertura de teste. O que as ferramentas de teste fazem é nada mais que executar o seu código e mapear as linhas que são executadas. Cabe a você, desenvolvedor ou desenvolvedora, garantir que seu teste se assemelha à forma como seu software é utilizado.

Para fechar esse tópico, vale lembrar que **você não precisa lutar para chegar em 100% de cobertura de testes no seu código o tempo todo**. Acabamos de ver um exemplo claro onde um percentual de cobertura não indica nada sobre a qualidade de um teste.

É normal que alguns trechos fiquem difíceis de testar em aplicações muito complexas. Isso acontece, mas mesmo assim, não se deixe levar por testes falhos ou que possam indicar coberturas que não foram, de fato, garantidas através de asserções. Saiba balancear o esforço que você tem escrevendo testes e a confiança que esse mesmo teste entrega a você.

4.7 Indo além nas configurações

Existem formas ainda mais práticas e avançadas de configurar o Jest sem ser apenas por argumentos através de sua CLI. Podemos criar um arquivo de configuração que será lido pelo Jest sempre que rodarmos um teste, de forma automática.

Por padrão, o Jest já procura um arquivo com nome `jest.config.js` na raiz do projeto. Vamos criá-lo exportando um objeto vazio:

```
// arquivo jest.config.js

module.exports = {
};
```

Para entender como esse arquivo de configuração pode ser útil, vamos apagar aquele comando `test:coverage` , que criamos anteriormente no nosso `package.json` :

```
"scripts": {
    "test": "jest",
    "test:watch": "jest --watch",
    "test:coverage": "jest --coverage" // vamos remover essa linha
},
```

Com isso não teremos mais o nosso relatório de cobertura. Agora vamos ajustar o arquivo `jest.config.js` para que essa configuração já fique por lá. Podemos fazer isso inserindo a chave `collectCoverage` no objeto que exportamos. Isso indicará ao Jest que, mesmo

executando os testes normalmente e sem o argumento `--coverage`, ele deve colher os dados sobre a cobertura e gerar o relatório para nós.

Vamos editar o arquivo `jest.config.js`:

```
// arquivo jest.config.js

module.exports = {
  collectCoverage: true,
};
```

E vamos rodar somente o comando `npm test` no terminal:

```
-------------------|---------|----------|---------|---------|-------------
------
File               | % Stmts | % Branch | % Funcs | % Lines | Uncovered
Line #s
-------------------|---------|----------|---------|---------|-------------
------
All files          |     100 |      100 |     100 |     100 |
 falso-positivo.js |     100 |      100 |     100 |     100 |
 operacoes.js      |     100 |      100 |     100 |     100 |
-------------------|---------|----------|---------|---------|-------------
------

Test Suites: 2 passed, 2 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.02 s
Ran all test suites.
```

Nosso relatório foi gerado! Faça um teste e mude o valor de `collectCoverage` para `false` e veja o resultado. Com isso, você verá que o relatório não será exibido.

Esse arquivo de configuração será muito importante ao trabalharmos com diversos ambientes de teste, como, por exemplo, testes para o front-end e para o back-end.

É possível, inclusive, instalar plugins para que o modo de assistência (`--watch`) do Jest fique ainda mais completo e nos ajude com diversos outros utilitários, através da opção `watchPlugins` .

Praticamente todas as configurações disponíveis pela CLI (e outras mais) podem ser configuradas através desse arquivo. A documentação oficial, disponível em https://jestjs.io/docs/pt-BR/configuration, lista várias outras opções que podemos configurar e é bem completa.

Se você preferir, também pode colocar essas configurações diretamente no seu `package.json` , através de um objeto `jest` , da seguinte forma:

```
"jest": {
    "collectCoverage": false
}
```

Mas nem sempre isso será algo que você vai querer, já que também é uma certa convenção de mercado deixar arquivos de configuração separados no projeto.

Algumas outras ferramentas de teste

Muitos outros projetos utilizam ferramentas diferentes que não são o próprio Jest.

Outros frameworks de teste são: Mocha https://mochajs.org/, Jasmine https://jasmine.github.io/, Chai https://www.chaijs.com/ e Karma https://karma-runner.github.io/latest/index.html. Essas são ferramentas bem completas e flexíveis para testes.

O pacote Sinon (https://sinonjs.org/) também é muito conhecido e realiza *mocks*/*fakes*/*spies*/*stubs* em testes, que são, basicamente, uma coleção de utilitários para simular funções e dados falsos. Veremos detalhadamente cada uma dessas utilidades ao longo dos testes conforme suas necessidades.

Embora o livro aborde apenas o Jest para realizar os testes de unidade e integração, essas outras ferramentas estão no mercado há muito mais tempo e também valem a pena terem seu espaço reconhecido.

Parte 2: Aplicando testes unitários em uma CLI

Chegou a hora de colocar a mão na massa e praticar toda a teoria que vimos!

Vamos fazer isso realizando os testes unitários de uma CLI (*command-line interface*, mais conhecida como ferramenta de linha de comando), que manipula dados de usuário.

Capítulo 5
Testando código síncrono

Assim como vimos o funcionamento da CLI do Jest, vamos trabalhar e realizar os testes da nossa própria CLI, para que possamos aprender testes unitários com uma aplicação mais simples e sem muitas camadas de abstração.

O projeto em que aplicaremos os testes está no repositório que contém os exemplos do livro (https://javascriptassertivo.com.br/). Trabalharemos em cima do conteúdo da pasta `projetos/02-aplicando-testes-unitarios-em-uma-cli` .

Esse diretório contém todo o projeto da aplicação CLI feita em Node. Antes de começarmos a testar, vamos dar uma rápida passada por ela e explicar como ela foi criada.

Diferente do CommonJS, que ainda é muito utilizado no Node em geral, essa aplicação (assim como as demais) foi criada utilizando ESModules, um padrão para se trabalhar com módulos em JavaScript.

Caso você queira entender mais sobre as diferenças entre os tipos de módulos e saber um pouco mais sobre essa história toda na linguagem JavaScript, o post disponível em

https://gabrieluizramos.com.br/modulos-em-javascript conta com algumas informações detalhadas sobre esse assunto.

No momento, a única coisa que precisamos saber é que, para trabalhar com módulos no NodeJS, precisamos utilizar o Node em alguma versão maior ou igual à v14.13.0, que é a atual enquanto escrevo este livro.

Após isso, para indicar que o nosso projeto será escrito utilizando esse módulo, é só inserir o seguinte conteúdo no `package.json`:

```
{
    "type": "module"
}
```

Isso já fará com que nossa aplicação possa trabalhar com esse novo padrão.

5.1 Como criar uma CLI em NodeJS

Antes de começarmos os testes, vamos entender como criar uma CLI utilizando Node.

Para fazer isso, você pode criar um outro projeto ou conferir todos esses arquivos e configurações direto no exemplo do repositório. Caso crie um novo projeto, não se esqueça de iniciar as configurações rodando o comando `npm init -y`.

Para iniciar, vamos criar um arquivo JavaScript qualquer, que rodará ao executarmos nossa CLI pelo terminal. Vamos chamá-lo de `cli.js`.

No começo desse arquivo, precisamos colocar uma instrução chamada `shebang`, que basicamente informa ao nosso terminal como aquele código deve ser executado. No caso, informaremos que ele deve ser executado utilizando o binário do Node. Essas instruções

`shebang` iniciam com `#!` e são seguidas pelo caminho do binário do arquivo do Node, em sua máquina.

Vamos inseri-la na primeira linha do arquivo `cli.js` da seguinte forma:

```
#!/usr/bin/env node
```

Vamos criar um `console.log` também, apenas para que tenhamos alguma saída no nosso terminal ao finalizar a criação da CLI:

```
#!/usr/bin/env node

console.log('Oi, CLI!');
```

Vamos configurar a execução desse arquivo. Fazemos isso inserindo a propriedade `bin` no arquivo `package.json`. Essa propriedade `bin` pode conter uma string ou um objeto:

```
{
    "bin": "cli.js",
    // ou
    "bin": {
        "nome-do-executavel": "cli.js"
    }
}
```

Ambas as opções servem para configurar um arquivo que será executado ao chamar o comando pela CLI. Caso informe uma string, o nome do comando será, por padrão, o nome do pacote informado na propriedade `name` do `package.json`. Por exemplo:

```
{
    "name": "seu-pacote",
    "bin": "arquivo.js"
}
```

Isso configurará um executável com `seu-pacote`.

Para configurar usando um objeto (permitindo até que você crie mais de um executável no mesmo projeto):

```
{
    "name": "seu-pacote",
    "bin": {
        "nome-do-executavel": "cli.js"
    }
}
```

Isso configurará `nome-do-executavel` (a chave informada no campo `bin`) como executável, o que permite um pouco mais de flexibilidade e, inclusive, configurar vários executáveis para um mesmo projeto.

No nosso caso, vamos configurar o valor de `bin` com um objeto, e o nome do nosso binário será `minha-primeira-cli-em-node`:

```
{
    "name": "seu-pacote",
    "bin": {
        "minha-primeira-cli-em-node": "cli.js"
    }
}
```

Agora, basta realizarmos o `link` desse projeto com as nossas configurações e `node_modules` (instalações de pacotes do Node) globais. Para fazer isso, basta estar na raiz do projeto e executar o seguinte comando no terminal:

```
npm link
```

Isso fará a instalação e o link do projeto. Então, basta executar, no terminal, o comando que configuramos:

```
minha-primeira-cli-em-node
```

Que veremos o seguinte resultado:

```
Oi, CLI!
```

Pronto! Bem prático. Caso você queira remover esse exemplo da sua máquina e não deixar esse binário configurado, basta executar o comando `npm unlink` no terminal.

5.2 Estrutura do projeto, instalação e configuração

Na raiz do projeto existe um arquivo `README.md` com algumas informações que valem a pena serem lidas. De qualquer forma, vamos dar uma olhada em como o projeto está estruturado e em como instalá-lo.

Com o projeto em sua máquina, primeiro veremos a estrutura de pastas e arquivos, para que possamos nos ambientar. Na raiz dele, existe um arquivo `cli.js`, que é basicamente o arquivo que inicia a aplicação através da CLI, importa a função principal e fornece os argumentos informados em sua execução. Também na raiz está localizado o arquivo `database.json`, que será utilizado para simularmos um banco de dados contendo os dados da aplicação.

A pasta `src` contém todo o código da aplicação. Embora não ocorra nenhum processo de transpilação/build, achei interessante manter essa estrutura por se tratar de um padrão adotado no mercado. Essa pasta, por sua vez, contém a seguinte estrutura:

- `index.js`: contém a função principal que inicia a aplicação;
- `constants`: diretório que contém as constantes utilizadas ao longo do projeto, que são, basicamente, as `roles.js` (diferentes níveis de usuário);
- `database`: contém a "camada" que realiza a manipulação do arquivo `database.json`, sendo:
  - `file.js`: faz a leitura/escrita do arquivo `database.json` propriamente dito;
  - `parser.js`: faz a formatação de dados para JSON/String e vice-versa;
  - `user`: diretório contendo as operações de `CRUD` de usuário (`create.js`, `read.js`, `remove.js`, `update.js`).
- `middlewares`: arquivos que funcionam como middlewares das operações, sendo:
  - `index.js`: que aplica a lógica da cadeia de middlewares;

    - `user.js` : middleware para validações de usuário (como validação de permissão);
    - `data.js` : middleware para validação do campo `data` em cada operação.
- `operations` : diretório que contém o manuseio das operações de `CRUD` , onde serão executadas através da CLI e retornarão mensagens ao usuário (camada intermediária entre a CLI e as operações do diretório `database`), por isso contém uma estrutura de arquivos bem semelhante à da pasta `database` ;
- `utils` : diretório que contém alguns utilitários necessários para a aplicação, como o de logging (`logger.js`) e o `args.js` , que formata/valida os argumentos enviados via terminal.

Alguns comandos utilitários (como o de criação da base de dados e o de gerar usuário) estão disponíveis na pasta `commands` , também na raiz do projeto, cada um em seu respectivo arquivo. Eles podem ser acessados através dos scripts do npm e, caso precise utilizá-los, eles são:

```
npm run db:create
# e
npm run user:generate
```

O primeiro comando executará o arquivo `commands/create-database.js` , que criará uma nova base de dados. O segundo simplesmente vai gerar o `JSON` de um usuário, para facilitar os testes locais (e também ajudar na nossa diversão com a CLI).

Para realizar os registros e facilitar a criação da base de dados, esse projeto usa um pacote chamado `faker` (https://www.npmjs.com/package/faker) na criação dos dados necessários.

Para instalar o projeto, basta estarmos em sua pasta e executar o seguinte comando no terminal:

```
npm i
```

Isso instalará todas as dependências do nosso projeto. Agora, vamos realizar o `link` como vimos anteriormente:

```
npm link
```

E a CLI `jsassertivo` estará disponível para ser executada!

Para trabalhar com essa CLI, é necessário informar quatro argumentos:

- `username` : contendo nome de um usuário para autenticação;
- `password` : contendo a senha desse mesmo usuário;
- `operation` : contendo uma string com alguma das quatro operações que podem ser realizadas (`create` , `read` , `remove` ou `update`);
- `data` : uma string de um `JSON` contendo os campos relacionados à operação que deseja realizar.

Todas as operações que envolvem escrita, como criação, remoção e atualização de um usuário, só podem ser feitas por outros usuários com privilégios de administradores, em outras palavras, apenas por usuários que possuam `role` do valor `ADMIN` ; qualquer outro usuário com a `role` padrão `USER` só poderá visualizar os dados dos demais usuários da base.

Pensando no usuário padrão que possui `username` e `password` com `admin` , os dois argumentos iniciais seriam:

```
jsasssertivo --username=admin --password=admin
```

Depois, basta informar a operação no campo `operation` :

```
jsassertivo --username=admin --password=admin --operation=read
```

E, no campo `data` , informar a string de um `JSON` relacionada à operação. Por exemplo, para ler os dados de um usuário com `uid` igual a `abc-123` , teríamos algo como:

```
jsassertivo --username=admin --password=admin --operation=read --
data='{"uid": "abc-123"}'
```

Para deletar um usuário, basta informar seu `uid` também:

```
# Agora com operation=remove
jsassertivo --username=admin --password=admin --operation=remove --
data='{"uid": "abc-123"}'
```

Já para fazer qualquer atualização em um cadastro, além de passar o `uid`, é só informar os novos campos com seus respectivos valores. Por exemplo, para atualizar o `name` de um usuário com o mesmo `uid` que comentamos:

```
# Agora com campos novos no data e operation=update
jsassertivo --username=admin --password=admin --operation=update --
data='{"uid": "abc-123", "name": "novoNome"}'
```

E para criar um usuário, basta informar um `JSON` contendo os campos de um novo cadastro: `userName`, `name`, `lastName`, `email`, `password`. O campo `avatar` pode conter a URL de uma foto para o perfil. Além disso, você pode informar uma `role` do tipo `ADMIN` ou `USER` para cadastrar e, caso não informe nenhuma, o novo registro terá a role `USER` por padrão:

```
# Informando operation=create e o JSON do novo usuário
jsassertivo --username=admin --password=admin --operation=update --
data='{"email":"Faustino_Hauck@hotmail.com","userName":"Bertrand59","passw
ord":"I6fsIfUOAdjFDt8","name":"Estell","lastName":"Armstrong"}'
```

5.3 Mão na massa: testando os utilitários de logging

Arrumando nosso ambiente e entendendo a estrutura inicial

A ideia é criarmos nossos testes na pasta `__tests__`, que também está na raiz do projeto. Por isso, você pode apagá-la para fazermos o passo a passo. Aproveite e também apague o arquivo `jest.config.js`, que também está na raiz do projeto, pois também

veremos o passo a passo de como criá-lo. Como evitaremos passar pelas implementações das funções (mas explicaremos o que elas fazem) é importante que você tome um tempo para analisar os códigos do projeto também.

Essa aplicação está utilizando os ESModules, então foi necessária uma pequena configuração para que os testes funcionassem corretamente. Foi necessário instalar um pacote chamado **Babel** (https://babeljs.io/), que é basicamente um "compilador" de JavaScript, que permite que utilizemos algumas funcionalidades mais modernas do JS e que se encarrega de compilar esse código mais "moderno" para uma versão mais "antiga". Além disso, foi instalado um pacote chamado `babel-jest`, que atua como a "ponte" entre o Babel, o Jest e um plugin do próprio Babel que converte os ESModules para CommonJS ao ser executado. Todos esses pacotes estão no `package.json`, na parte de `devDependencies`.

Para configurar tudo isso junto, a única coisa necessária foi criar um arquivo `.babelrc` contendo esse JSON de configuração:

```
{
  "plugins": ["transform-es2015-modules-commonjs"]
}
```

Esse arquivo também está na raiz do projeto. Toda essa configuração do Babel já foi realizada, então não precisa se preocupar em fazê-la, mas é necessário falar sobre ela caso você precise configurar em algum outro projeto também. O Jest também já está instalado, então vamos aos testes!

Vamos começar por testar os arquivos da pasta `utils`, mais especificamente as funções do arquivo `logger`. Esse arquivo basicamente exporta um objeto com três funções: `log`, `success` e `error`, além de um objeto `COLORS`, que contém as chaves `DEFAULT`, `GREEN` e `RED`, utilizadas para customizar esses logs com mensagens coloridas.

Na pasta `__tests__`, vamos criar uma estrutura semelhante à pasta `src`, criando um novo diretório chamado `utils` e um novo arquivo chamado `logger.test.js` dentro dele. Após isso, vamos importar o utilitário de log da seguinte forma:

```
// arquivo __tests__/utils/logger.test.js
import logger from '../../src/utils/logger.js';
```

Iniciando os testes

Vamos iniciar os testes para esse arquivo. Tudo o que precisamos fazer nessas simples funções é basicamente chamá-las e verificar se elas chamaram o `console.log` com os argumentos corretos. Entretanto, como vamos saber se a função `console.log` foi chamada?

Podemos fazer isso utilizando o `spyOn` do Jest, uma função muito importante que encapsula alguma outra função qualquer e permite que realizemos asserções em cima dela. Para a executarmos, basta chamar `spyOn` e informar um objeto como primeiro parâmetro e também uma chave desse mesmo objeto para que ela seja observada. Vamos ver como utilizá-la no nosso exemplo:

```
// arquivo __tests__/utils/logger.test.js
import logger  from '../../src/utils/logger.js';

// criamos spy para verificar a função log dentro de console
const spyLog = jest.spyOn(console, 'log');
```

Isso nos retornará uma função do tipo `jest.fn()`, que é o padrão para todas as funções que o Jest cria para facilitar nossos testes. Essas funções são mais conhecidas pelo termo *mock* e nada mais são do que funções que podemos controlar de dentro de nossos testes controlando seu comportamento, suas implementações e também realizando algumas asserções.

Podemos pensar nos mocks como os dublês dos filmes a que assistimos por aí. Os dublês fazem as cenas de ação em um filme,

mas quem fica com a fama no final é a pessoa que eles estão interpretando (fingindo ser), certo? Nosso código funcionará do mesmo jeito! Nossas funções vão interagir com outras funções ou módulos quando nossa aplicação estiver realmente funcionando, mas dentro do nosso ambiente de teste quem vai entrar em ação é um dublê que poderá simular essas determinadas ações e permitirá que controlemos o que deve acontecer (como cenários de sucesso e falha).

Portanto, de agora em diante, vamos nos referir a essas funções do Jest como mocks, ok? Então podemos criar nossos casos de testes propriamente ditos:

```
import logger from '../../src/utils/logger.js';

const spyLog = jest.spyOn(console, 'log');

it('Funções de logging: log', () => {
  // chamamos função de log normal
  logger.log('teste');

  // verificamos se o spy foi chamado
  expect(spyLog).toHaveBeenCalledTimes(1);
});
```

Vamos rodar o comando `npm run test:watch` para ficarmos assistindo às mudanças no teste conforme vamos alterando os arquivos. Com isso, você verá que o primeiro teste passou:

```
 PASS  __tests__/utils/logger.test.js
  ? Funções de logging: log (16 ms)

  console.log
     teste

      at CustomConsole.<anonymous> (node_modules/jest-
mock/build/index.js:814:25)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
```

```
Snapshots:   0 total
Time:        1.137 s
```

Show! Vamos fazer a mesma coisa para as funções de `success`:

```
import logger from '../../src/utils/logger.js';

const spyLog = jest.spyOn(console, 'log');

it('Funções de logging: log', () => {
  // chamamos função de log normal
  logger.log('teste');

  // verificamos se o spy foi chamado
  expect(spyLog).toHaveBeenCalledTimes(1);
});

it('Funções de logging: success', () => {
  // chamamos função de log success
  logger.success('teste');

  // verificamos se o spy foi chamado
  expect(spyLog).toHaveBeenCalledTimes(1);
});
```

Após salvar o arquivo, você notará que temos um erro:

```
  ? Funções de logging: success

    expect(jest.fn()).toHaveBeenCalledTimes(expected)

    Expected number of calls: 1
    Received number of calls: 2
```

Estranho, não? O que está acontecendo é que, após a execução do primeiro teste, nossa variável `spyLog` foi chamada uma vez. Como a função `logger.success` também executa a função `console.log`, nós estamos acumulando essa quantidade de chamadas dentro da variável que criamos.

Com isso, vemos um fundamento muito importante: o de limpar nossos mocks e evitar um certo "estado" entre os testes.

Para fazer isso, podemos chamar a função `mockClear`, que também é fornecida pelo Jest, dentro do nosso segundo teste. Essa função limpa as chamadas de qualquer função do tipo `jest.fn()`. Fazemos isso da seguinte maneira:

```
import logger from '../../src/utils/logger.js';

const spyLog = jest.spyOn(console, 'log');

it('Funções de logging: log', () => {
  logger.log('teste');
  expect(spyLog).toHaveBeenCalledTimes(1);
});

it('Funções de logging: success', () => {
  // adicionamos a linha abaixo
  spyLog.mockClear();

  logger.success('teste');
  expect(spyLog).toHaveBeenCalledTimes(1);
});
```

E assim, nosso teste passa! Entretanto, caso tivéssemos mais funções que executam `console.log`, teríamos que lembrar de ficar executando `spyLog.mockClear()` e isso não é algo que podemos garantir que lembraremos. Seria bem bacana automatizar essa "limpeza", não acha?

Adicionando os hooks

Se você lembrou dos hooks, que comentamos no capítulo anterior, podemos justamente aplicá-los nesse cenário. Temos quatro hooks disponíveis para utilizarmos: `beforeEach`, `beforeAll`, `afterEach` e `afterAll`. Como queremos realizar essa limpeza antes da execução de cada teste, podemos utilizar o hook `beforeEach`.

Vamos colocá-lo antes de qualquer um dos nossos testes, executando a função `mockClear`, que fizemos, e removendo-a do segundo teste:

```
import logger from '../../src/utils/logger.js';

const spyLog = jest.spyOn(console, 'log');

// criamos beforeEach que executa o mockClear
beforeEach(() => {
  spyLog.mockClear();
});

it('Funções de logging: log', () => {
  logger.log('teste');
  expect(spyLog).toHaveBeenCalledTimes(1);
});

it('Funções de logging: success', () => {
  // removemos mockClear
  logger.success('teste');
  expect(spyLog).toHaveBeenCalledTimes(1);
});
```

Ao salvar o arquivo, você verá que tudo funciona como deveria! Faça uma mudança para ver que nosso teste pode falar: mude as execuções de `console.log` do arquivo `src/utils/logger.js` para executar `console.info` e você verá que os testes falharão.

Se você olhou com atenção para os resultados dos testes, você deve ter percebido que, como nossas funções executam `console.log`, várias mensagens estão aparecendo no terminal. Podemos remover a implementação do `console.log` utilizando nosso `spyLog` para que não tenhamos essa saída tão poluída. Para fazer isso, podemos utilizar a função `mockImplementation`, que também é padrão de qualquer função do tipo `jest.fn()`, da seguinte maneira:

```
import logger from '../../src/utils/logger.js';

// adicionamos .mockImplementation()
```

```
const spyLog = jest.spyOn(console, 'log').mockImplementation();

beforeEach(() => {
  spyLog.mockClear();
});

it('Funções de logging: log', () => {
  logger.log('teste');
  expect(spyLog).toHaveBeenCalledTimes(1);
});

it('Funções de logging: success', () => {
  logger.success('teste');
  expect(spyLog).toHaveBeenCalledTimes(1);
});
```

O que essa função faz é, basicamente, sobrescrever a implementação original de alguma função `jest.fn()` (que é retornada pelo `spyOn`). Com isso, nossos logs não aparecerão no teste. Podemos, inclusive, fornecer qualquer implementação para a função `mockImplementation` . Vamos fazer a seguinte mudança:

```
// passamos uma função para mockImplementation
const spyLog = jest.spyOn(console, 'log').mockImplementation((...args) =>
{
  console.info('mockImplementation');
  console.info(...args);
});
```

Ao salvar esse novo ajuste, perceberemos que a nossa função dentro de `mockImplementation` foi executada! Isso nos possibilita fazer várias coisas nos nossos testes. Vamos voltar o `mockImplementation` do jeito que estava (sem passar uma função).

Entretanto, é interessante remover essa implementação vazia após os nossos testes. Isso é necessário pois, caso algum outro teste dentro do mesmo arquivo usasse o `console.log` original, a função não surtiria seu efeito, por exemplo:

```
const spyLog = jest.spyOn(console, 'log').mockImplementation();

beforeEach(() => {
  spyLog.mockClear();
});

it('Funções de logging: log', () => {
  logger.log('teste');
  expect(spyLog).toHaveBeenCalledTimes(1);
});

it('Funções de logging: success', () => {
  logger.success('teste');
  expect(spyLog).toHaveBeenCalledTimes(1);
});

// linha com console.log que não será exibido
console.log('isso não funciona :(');
```

Para fazer isso, vamos utilizar outro hook que executa ao final de todos os testes: `afterAll`. Ele será responsável por executar a função `jest.restoreAllMocks()`, que restaura todas as implementações feitas anteriormente:

```
beforeEach(() => {
  spyLog.mockClear();
});

// inserimos esse after all com jest.restoreAllMocks() e alguns logs para testar
afterAll(() => {
  console.log('não funciona');
  jest.restoreAllMocks();
  console.log('isso funciona');
});

it('Funções de logging: log', () => {
  logger.log('teste');
  expect(spyLog).toHaveBeenCalledTimes(1);
});
```

```
it('Funções de logging: success', () => {
  logger.success('teste');
  expect(spyLog).toHaveBeenCalledTimes(1);
});
```

Ao salvar esse arquivo, você verá que o último log com o texto `isso funciona` aparecerá. Após isso, podemos remover esses `console.log` que criamos. Nossos testes para o arquivo `logger` já estão bem bacanas. Para que possamos aprimorar ainda mais, proponho um desafio a você: crie testes para a função `logger.error` , que executa o `console.error` . O processo será exatamente o mesmo que fizemos até agora.

Após isso, agrupe tudo em uma função `describe` para organizar e melhorar os títulos de cada teste da maneira que você achar mais interessante.

Dica: para facilitar a limpeza das chamadas dos mocks e limpar todos, e não somente o `spyLog` , você pode utilizar o `jest.clearAllMocks()` no hook `beforeEach` , em vez de ficar limpando cada mock manualmente.

5.4 Testando o utilitário de argumentos

Vamos fazer o teste para as funções exportadas pelo arquivo `src/utils/args.js` . Para isso, vamos criar o arquivo `args.test.js` na pasta `__tests__/utils` . Começaremos importando as duas funções exportadas por esse arquivo: `parse` (exportada como `default`) e `args` .

```
// arquivo __tests__/utils/args.test.js
import parse, { validateArgs } from '../../src/utils/args.js';
```

Vamos começar testando a função `parse` . O que ela faz basicamente é pegar os argumentos fornecidos pelo terminal no modelo `--argumento=valor` e transformá-los em um objeto para que as

funções possam trabalhar com os dados de uma forma mais conveniente. Ou seja, todas as informações que passamos da seguinte maneira ao executar a CLI:

```
jsassertivo --username=admin --password=admin --operation=operacao --
data='{"uid": "abc-123"}'
```

Tornam-se o seguinte objeto:

```
{
   "username": "admin",
   "password": "admin",
   "operation": "operacao",
   "data": {
       "uid": "abc-123"
   }
}
```

No entanto, ao olhar a assinatura dessa função, podemos ver os seguintes parâmetros:

```
export default function parse (_env, _bin, ...cliArgs)
```

Os dois primeiros parâmetros não são utilizados. Isso acontece porque, ao executar uma CLI, todos os argumentos fornecidos ao comando são acessados em um formato de array mais ou menos assim:

```
[
  '/Users/gabriel.ramos/.nvm/versions/node/v14.11.0/bin/node',
  '/Users/gabriel.ramos/.nvm/versions/node/v14.11.0/bin/jsassertivo',
  '--username=admin',
  '--password=admin',
  '--operation=opercao',
  '--data={"uid": "abc-123"}'
]
```

Se olharmos com atenção, as duas primeiras posições desse array são caminhos do binário do Node (no caso, as informações são da minha máquina) e do binário da própria CLI que está sendo executada. Já os argumentos seguintes são os dados informados ao

executar a CLI. Isso quer dizer que, para a função de `parse` funcionar, os dois primeiros argumentos são desnecessários.

Já temos uma ideia de que, para fazer o teste dessa função, precisaremos simular esse dado corretamente. Com isso, vamos começar criando um primeiro teste e uma variável que pode, inclusive, conter o array acima para nos ajudar.

```
// arquivo __tests__/utils/args.test.js
import parse, { validateArgs } from '../../src/utils/args.js';

// criamos o teste com a variável argumentos
it('Faz o parse dos argumentos da CLI', () => {
  const argumentos = [
    '/Users/gabriel.ramos/.nvm/versions/node/v14.11.0/bin/node',
    '/Users/gabriel.ramos/.nvm/versions/node/v14.11.0/bin/jsassertivo',
    '--username=admin',
    '--password=admin',
    '--operation=operacao',
    '--data={"uid": "abc-123"}'
  ];
});
```

Vamos criar um objeto contendo o retorno que já sabemos que a função deve nos devolver. Pode copiar o objeto que comentamos anteriormente também.

```
import parse, { validateArgs } from '../../src/utils/args.js';

it('Faz o parse dos argumentos da CLI', () => {
  const argumentos = [
    '/Users/gabriel.ramos/.nvm/versions/node/v14.11.0/bin/node',
    '/Users/gabriel.ramos/.nvm/versions/node/v14.11.0/bin/jsassertivo',
    '--username=admin',
    '--password=admin',
    '--operation=operacao',
    '--data={"uid": "abc-123"}'
  ];

  // agora com os dados formatados
```

```
  const dados = {
    username: "admin",
    password: "admin",
    operation: "operacao",
    data: {
      uid: "abc-123"
    }
 };
});
```

Podemos criar nossa asserção executando a função `parse` e fornecendo os argumentos, e verificar se o objeto de retorno é igual ao `dados`.

```
import parse, { validateArgs } from '../../src/utils/args.js';

it('Faz o parse dos argumentos da CLI', () => {
  const argumentos = [
    '/Users/gabriel.ramos/.nvm/versions/node/v14.11.0/bin/node',
    '/Users/gabriel.ramos/.nvm/versions/node/v14.11.0/bin/jsassertivo',
    '--username=admin',
    '--password=admin',
    '--operation=operacao',
    '--data={"uid": "abc-123"}'
  ];

  const dados = {
    username: "admin",
    password: "admin",
    operation: "operacao",
    data: {
      uid: "abc-123"
    }
 };

 const retornado = parse(argumentos);

 expect(retornado).toEqual(dados);
});
```

Vamos testar a função `validateArgs`. Essa função recebe dois parâmetros, o primeiro é um objeto com quaisquer campos, e o segundo, um array utilizado para validar esse mesmo objeto, ou seja, ele verifica se cada string do array informado é uma chave no objeto.

Tanto o primeiro quanto o segundo parâmetro possuem valores padrões atribuídos: o primeiro tem um valor padrão de um objeto vazio `{}`, e o segundo, um array que contém os valores `username`, `password`, `operation` e `data`. Isso nada mais é do que a validação padrão da CLI caso você esqueça de passar algum desses parâmetros ou digite errado.

Todos os retornos dessa validação são baseados em um objeto contendo uma chave `valid`, que retorna um `booleano` que diz se o valor é válido ou não, e uma chave `message`, que, caso o valor de `valid` seja `falso`, conterá uma mensagem com o erro.

Ou seja, os casos que precisamos cobrir de testes são os seguintes:

- Uma operação válida, onde todos os campos no objeto são existentes;
- Uma operação válida onde informamos somente o objeto e os valores dos campos são os padrões;
- Uma operação inválida onde passamos um objeto sem valores nenhum e, consequentemente, recebemos uma mensagem de erro;
- Uma operação inválida onde passamos um objeto com chaves diferentes dos campos validados, também resultando em uma mensagem de erro.

Vamos ao primeiro cenário. Ainda no arquivo `__tests__/utils/args.test.js`, vamos criar um *describe* para agrupar todos esses testes e iniciar o teste para o cenário de sucesso:

```
// arquivo __tests__/utils/args.test.js
// restante do código omitido
```

```
// criamos describe
describe('Validação de argumentos da CLI', () => {
  it('Valida com sucesso os campos informados', () => {
    // e um it vazio
  });
});
```

Vamos desenvolver o teste para o cenário de sucesso propriamente dito montando um objeto, como o utilizado no teste anterior:

```
describe('Validação de argumentos da CLI', () => {
  it('Valida com sucesso os campos informados', () => {
    // variavel dados
    const dados = {
      username: "admin",
      password: "admin",
      operation: "operacao",
      data: {
        uid: "abc-123"
      }
    };
  });
});
```

E um array de campos a serem validados contendo os campos do objeto:

```
describe('Validação de argumentos da CLI', () => {
  it('Valida com sucesso os campos informados', () => {
    const dados = {
      username: "admin",
      password: "admin",
      operation: "operacao",
      data: {
        uid: "abc-123"
      }
    };

    // variável campos
    const campos = ['username', 'password', 'operation', 'data'];
```

```
  });
});
```

Vamos criar nossa variável contendo o retorno da função `validateArgs` e verificar que ela retorna um objeto com `valid` igual a `true` e `message` sendo uma `string` vazia:

```
describe('Validação de argumentos da CLI', () => {
  it('Valida com sucesso os campos informados', () => {
    const dados = {
      username: "admin",
      password: "admin",
      operation: "operacao",
      data: {
        uid: "abc-123"
      }
    };

    const campos = ['username', 'password', 'operation', 'data'];

    // variável de retorno
    const retornado = validateArgs(dados, campos);

    // asserção
    expect(retornado).toEqual({ valid: true, message: '' });
  });
});
```

Como somente o `valid` nos interessa nesse caso, podemos verificar somente se o valor dele é igual a `true` fazendo a seguinte alteração:

```
describe('Validação de argumentos da CLI', () => {
  it('Valida com sucesso os campos informados', () => {
    const dados = {
      username: "admin",
      password: "admin",
      operation: "operacao",
      data: {
        uid: "abc-123"
      }
    };
```

```
    const campos = ['username', 'password', 'operation', 'data'];

    const retornado = validateArgs(dados, campos);

    // alteramos a asserção para verificar somente o retorno de valid
    expect(retornado.valid).toEqual(true);
  });
});
```

Podemos utilizar esse mesmo teste para validar o segundo caso, onde não informamos os campos. Vamos criar uma nova asserção:

```
describe('Validação de argumentos da CLI', () => {
  it('Valida com sucesso os campos informados', () => {
    const dados = {
      username: "admin",
      password: "admin",
      operation: "operacao",
      data: {
        uid: "abc-123"
      }
    };

    const campos = ['username', 'password', 'operation', 'data'];

    const retornado = validateArgs(dados, campos);

    expect(retornado.valid).toEqual(true);
    expect(validateArgs(dados).valid).toEqual(true); // nova asserção
  });
});
```

Caso queiramos manter um certo padrão e deixar a execução da função `validateArgs` diretamente no `expect`, podemos modificar a asserção anterior também:

```
describe('Validação de argumentos da CLI', () => {
  it('Valida com sucesso os campos informados', () => {
    const dados = {
      username: "admin",
```

```
      password: "admin",
      operation: "operacao",
      data: {
        uid: "abc-123"
      }
    };

    const campos = ['username', 'password', 'operation', 'data'];

    // apagamos a variável retornada e modificamos a asserção abaixo
    expect(validateArgs(dados, campos).valid).toEqual(true);
    expect(validateArgs(dados).valid).toEqual(true);
  });
});
```

Se você olhou atentamente, percebeu que podemos reutilizar nossa variável `dados` do último teste. Atualmente, nosso arquivo está assim:

```
import parse, { validateArgs } from '../../src/utils/args.js';

it('Faz o parse dos argumentos da CLI', () => {
  const argumentos = [
    '/Users/gabriel.ramos/.nvm/versions/node/v14.11.0/bin/node',
    '/Users/gabriel.ramos/.nvm/versions/node/v14.11.0/bin/jsassertivo',
    '--username=admin',
    '--password=admin',
    '--operation=operacao',
    '--data={"uid": "abc-123"}'
  ];

  const dados = {
    username: "admin",
    password: "admin",
    operation: "operacao",
    data: {
      uid: "abc-123"
    }
  };

 const retornado = parse(argumentos);
```

```
 expect(retornado).toEqual(dados);
});

describe('Validação de argumentos da CLI', () => {
  it('Valida com sucesso os campos informados', () => {
    const dados = {
      username: "admin",
      password: "admin",
      operation: "operacao",
      data: {
        uid: "abc-123"
      }
    };

    const campos = ['username', 'password', 'operation', 'data'];

    expect(validateArgs(dados, campos).valid).toEqual(true);
    expect(validateArgs(dados).valid).toEqual(true);
  });
});
```

Podemos colocar nossa variável `dado` fora do escopo de ambos os testes, assim os dois cenários (e os próximos) podem utilizá-la. Vamos ajustar:

```
import parse, { validateArgs } from '../../src/utils/args.js';

// variável agora é acessível para todos os testes
const dados = {
  username: "admin",
  password: "admin",
  operation: "operacao",
  data: {
    uid: "abc-123"
  }
};

it('Faz o parse dos argumentos da CLI', () => {
  const argumentos = [
    '/Users/gabriel.ramos/.nvm/versions/node/v14.11.0/bin/node',
```

```
    '/Users/gabriel.ramos/.nvm/versions/node/v14.11.0/bin/jsassertivo',
    '--username=admin',
    '--password=admin',
    '--operation=operacao',
    '--data={"uid": "abc-123"}'
  ];
  // removemos dados daqui
  const retornado = parse(argumentos);

  expect(retornado).toEqual(dados);
});

describe('Validação de argumentos da CLI', () => {
  it('Valida com sucesso os campos informados', () => {
    // removemos daqui também

    const campos = ['username', 'password', 'operation', 'data'];

    expect(validateArgs(dados, campos).valid).toEqual(true);
    expect(validateArgs(dados).valid).toEqual(true);
  });
});
```

Esse trabalho de olhar para as variáveis e tentar reutilizar trechos de código pode nos ajudar bastante. Vale sempre lembrar que um teste também é código, então você pode criar qualquer abstração necessária no JavaScript para ajudar.

Por último, vamos fazer os cenários de falha, nos quais não passamos nenhum objeto ou os campos não são iguais. Vamos criar um novo teste já com a primeira asserção executando `validateArgs` sem informar nenhum argumento e também precisaremos validar o campo `mensagem` :

```
describe('Validação de argumentos da CLI', () => {
  it('Valida com sucesso os campos informados', () => {
    const campos = ['username', 'password', 'operation', 'data'];

    expect(validateArgs(dados, campos).valid).toEqual(true);
    expect(validateArgs(dados).valid).toEqual(true);
```

```
  });

  // novo teste com asserção
  it('Valida os cenários de erro e retorna uma mensagem', () => {
    expect(validateArgs()).toEqual({
      valid: false,
      message: ''
    })
  });
});
```

No entanto, ao salvar esse arquivo, teremos um teste, pois agora existirá uma mensagem no campo `message`. Caso queiramos verificar que o valor da mensagem é exatamente igual ao retornado (e ao que esperamos), temos duas alternativas:

- Colocar a mensagem diretamente no campo `message`;
- Exportar a função que cria a mensagem para que possamos utilizá-la e facilitar nosso teste.

Caso não queiramos validar a mensagem, mas apenas verificar que nos foi retornada uma `string`, podemos utilizar o `expect.any`, uma função que aceita qualquer construtor e valida se algum dado foi gerado a partir desse construtor, que, no nosso caso, é `String`.

Vamos optar por esse cenário para aplicar o `expect.any`:

```
it('Valida os cenários de erro e retorna uma mensagem', () => {
  expect(validateArgs()).toEqual({
    valid: false,
    message: expect.any(String)
  });
});
```

Precisamos validar mais um cenário de erro em que o objeto e os campos são diferentes. Nesse mesmo teste, vamos fazer mais uma asserção e, utilizando a variável `dados`, vamos passar uma lista de campos quaisquer para nosso teste nos retornar outro erro:

```
it('Valida os cenários de erro e retorna uma mensagem', () => {
    expect(validateArgs()).toEqual({
      valid: false,
      message: expect.any(String)
    });
    // nova asserção
    expect(validateArgs(dados, ['email'])).toEqual({
      valid: false,
      message: expect.any(String)
    });
});
```

Vamos gerar nosso relatório de erro com o Jest para ver como estão nossos testes. Rodando `npm run test:coverage`, você poderá ver o relatório no terminal ou através do arquivo HTML dentro de `coverage/lcov-report/index.html`. Caso não tenhamos batido 100% de cobertura, não precisamos nos preocupar, pois nossos testes estão cobrindo os cenários necessários.

Talvez você esteja se perguntando: mas e as outras funções do arquivo? Por que elas aparecem como testadas mesmo que não tenhamos feito testes específicos para elas? A resposta é que, como elas são utilizadas pelas duas funções principais que testamos, automaticamente já estamos garantindo o comportamento delas. De certa forma, podemos olhar isso como se fosse um início bem embrionário de um teste de integração.

Ao final, caso qualquer uma dessas funções internas mude e altere o comportamento das funções principais, nosso teste também nos alertará. Já temos bastante segurança e confiança no que fizemos até aqui.

Com isso, aproveito para já deixar outro desafio para você: exportar a função que gera a mensagem e aplicá-la no teste!

Nem só de código síncrono vive uma aplicação

Entendemos como aplicar de forma efetiva testes em funções síncronas, e chegou a hora de entender como funções assíncronas

devem ser testadas.

Capítulo 6
Testando código assíncrono

Até o momento, só testamos códigos síncronos: códigos que são executados exatamente na ordem em que são definidos e não dependem de nada mais complexo, como uma requisição a um serviço, salvamento de um arquivo, utilização de um temporizador. No entanto, nem só de código síncrono vive uma aplicação JavaScript, certo?

Agora veremos como aplicar testes em funções assíncronas ao testar a nossa camada que salva os dados da CLI, com os respectivos cenários de sucesso e erro.

6.1 Hora das funções assíncronas: testando a camada que salva os dados

Testando a criação de usuários

Vamos começar pelo arquivo `src/database/user/create.js`, onde está a função principal que cadastra um usuário no arquivo `database.json`.

O arquivo `src/database/user/create.js` importa o `faker` para gerar dados automáticos (como o `uid`) e importa as `ROLES` de usuário para realizar o cadastro com algum valor. Ele também importa as funções `loadDatabase` e `saveDatabase`, que estão no arquivo `src/database/file.js`, que é responsável por manipular o arquivo que usamos para simular a base de dados.

Dentro da pasta `__tests__`, vamos criar outra pasta, `database/user`, com o arquivo `create.test.js`. Após isso, vamos importar a função diretamente do arquivo que vamos testar.

```
// arquivo __tests__/database/user/create.test.js
import { createUser } from '../../../src/database/user/create.js';
```

Em um primeiro momento, é tentador olhar para a função e simplesmente escrever um teste e chamá-la, certo? No entanto, temos que ter um pouco mais de cuidado agora. Isso é necessário porque essa função, além de lidar com código assíncrono, diferente da função que testamos anteriormente, tem um efeito colateral: ela modifica um arquivo.

Quando lidamos com código assíncrono ou qualquer código que tenha algum efeito colateral, como chamar uma API, salvar algum dado em um banco ou enviar alguma informação a uma fila, na maioria das vezes não queremos que essa ação seja realizada. Não queremos modificar nosso arquivo `database.json` a cada vez que nossos testes rodarem e, nos outros exemplos, não seria uma boa prática ficar chamando alguma API ou salvando algum dado localmente apenas para executar os testes.

Precisamos realizar um mock das funções que salvam os dados no arquivo de registro, para que elas não sejam executadas e também para que possamos simular seus comportamentos. É algo bem parecido com o que fizemos no exercício anterior com a função `console.log`. Poderíamos simplesmente utilizar o `spyOn` da seguinte maneira:

```
import * as file from '../../../src/database/file.js';

const spies = {
  load: jest.spyOn(file, 'loadDatabase'),
  save: jest.spyOn(file, 'saveDatabase'),
};
```

Com o que aprendemos anteriormente, utilizando o `mockImplementation`, conseguiríamos chegar ao resultado que queremos e simular o comportamento dessas funções. Entretanto, gostaria de comentar sobre uma outra função chamada `jest.mock`. O que essa função faz é, basicamente, realizar o mock (mais

conhecido como "mockar") de todas as funções de algum módulo, de forma automática. Todos os mocks do Jest são aquelas funções do tipo `jest.fn()`, que já vimos.

O `jest.mock` recebe um argumento (obrigatoriamente) que é o caminho do módulo para realizar o mock. O segundo argumento é uma função opcional que, caso seja informada, deve retornar os valores "mockados" do arquivo.

Para entendermos melhor o que estou dizendo, vamos importar o módulo `file` inteiro e vamos fazer um `console.log` dele, executando a função `jest.mock`:

```
import { createUser } from '../../../src/database/user/create.js';
import * as file from '../../../src/database/file.js';

// adicionamos o mock
jest.mock('../../../src/database/file.js');
jest.mock('../../../src/database/path.js', () => null);

it('Cria usuário corretamente', () => {
  console.log(file);
});
```

Ao olhar o log no terminal (após rodar `npm run test:watch`) você verá que todas as funções do arquivo `file.js` que foram exibidas no `console.log` foram sobrescritas com funções mock do Jest!

Eu sei, você notou que tem um `jest.mock` extra aí! Isso é necessário pois o Jest ainda não suporta muito bem os ESModules (lembra que instalamos um plugin do Babel para nos ajudar com isso?). Existe até uma Issue do GitHub fechada sobre isso (https://github.com/facebook/jest/issues/9213). Então, infelizmente precisamos "mockar" o retorno desse arquivo.

Para que não precisemos ficar informando que ele deve retornar `null` e até para facilitar testes futuros, podemos realizar o mock dele de uma maneira diferente: através da pasta `__mocks__`. Essa pasta é uma pasta padrão do Jest que, caso exista, pode servir para

realizar um mock predefinido sem que haja a necessidade de ficar passando a função como segundo parâmetro.

Olhe com atenção o arquivo na pasta `src/database/__mocks__/path.js`. Tudo o que ele contém é:

```
export default null;
```

Vamos trocar esse `null` para `1234`:

```
export default 1234;
```

No nosso teste, vamos apagar a função que passamos como segundo parâmetro para o mock do arquivo `path`, também vamos importar o arquivo `path` e fazer um `console.log` dele.

```
import { createUser } from '../../../src/database/user/create.js';
import * as file from '../../../src/database/file.js';
// importamos path
import path from '../../../src/database/path.js';

jest.mock('../../../src/database/file.js');
// apagamos a função passada como segundo argumento
jest.mock('../../../src/database/path.js');

it('Cria usuário corretamente', () => {
  console.log(path)
  console.log(file);
});
```

Com isso, você verá que `1234` aparecerá no log e que nosso mock ainda funciona! Apenas reutilizamos um valor padrão para que não precisemos informar toda vez que executarmos uma função `jest.mock`. Com isso, podemos apagar ambos os logs e começar nosso teste.

A função `createUser` que vamos testar recebe um objeto com cinco campos: `userName`, `password`, `email`, `name`, `lastName`. A função também pode receber uma `role`, mas, caso não seja informada,

cadastrará o usuário como `USER` por padrão. Essa função também retorna os dados do usuário criado.

Com isso, vamos criar uma variável de mock contendo os dados fictícios de um usuário e vamos começar nosso teste:

```
import { createUser } from '../../../src/database/user/create.js';
import * as file from '../../../src/database/file.js';

jest.mock('../../../src/database/file.js');
jest.mock('../../../src/database/path.js');

const usuario = {
  email: 'qualquer@email.com',
  password: 'senha1234',
  userName: 'usuárioQualquer',
  name: 'Usuário',
  lastName: 'Qualquer'
};

it('Cria usuário corretamente', () => {
});
```

Vamos executar a função `createUser` informando esse usuário como parâmetro e pegar seu usuário criado como retorno:

```
import { createUser } from '../../../src/database/user/create.js';
import * as file from '../../../src/database/file.js';

jest.mock('../../../src/database/file.js');
jest.mock('../../../src/database/path.js');

const usuario = {
  email: 'qualquer@email.com',
  password: 'senha1234',
  userName: 'usuarioQualquer',
  name: 'Usuário',
  lastName: 'Qualquer'
};

it('Cria usuário corretamente', () => {
```

```
  const user = createUser(usuario);
});
```

Ao salvar e rodar os testes, você perceberá que, mesmo com eles passando ("magicamente"), temos um erro:

```
UnhandledPromiseRejectionWarning: TypeError: Cannot read property 'push'
of undefined
```

Isso aconteceu porque, embora não tenhamos feito nenhuma asserção, o código que estamos testando utiliza `Promises` e é assíncrono. Esse erro está nos indicando justamente que não lidamos com uma Promise que foi rejeitada e que não foi possível executar o `push` que é feito dentro da nossa função.

Ou seja, a primeira coisa que precisamos fazer é adicionar `async/await` no nosso teste. Colocando `async` na função do teste e `await` ao executar `createUser`, temos:

```
it('Cria usuário corretamente', async () => {
  const user = await createUser(usuario);
});
```

Como a função `loadDatabase` retorna um array de usuários, justamente o que faremos para salvar um novo é adicionar um novo registro nesse array e salvamos seu valor final no arquivo.

Realizando o mock do array de usuários

Para que tenhamos esse array, precisamos fazer com que a função `loadDatabase`, que está mockada, retorne um array. Podemos fazer isso com a função `mockImplementation` também. Como essa função também retorna uma Promise, teremos que retornar essa Promise da nossa implementação, que será resolvida para um array da seguinte forma:

```
it('Cria usuário corretamente', async () => {
  file.loadDatabase.mockImplementation(() => Promise.resolve([]));
  const user = await createUser(usuario);
});
```

Com isso, o erro sumirá, mas não se engane. Ainda não fizemos nenhuma asserção e também não garantimos que as funções de carregamento/salvamento foram executadas corretamente.

Nossa função `mockImplementation` ficou um pouco verbosa. Como sabemos que ela retorna uma Promise, podemos utilizar uma outra função chamada `mockResolvedValue` e simplesmente passar nosso array como resultado da Promise resolvida:

```
it('Cria usuário corretamente', async () => {
  // modificamos mockImplementation por mockResolvedValue
  file.loadDatabase.mockResolvedValue([]);
  const user = await createUser(usuario);
});
```

Bem mais prático e legível, não? O Jest expõe a função `mockRejectedValue` também, para lidar com os casos de Promises rejeitadas funcionando da mesma maneira.

Tanto essas duas funções, como a função `mockImplementation`, podem ser usadas nas suas versões mais "efêmeras" que possuem o sufixo `Once` (que significa "única" ou "único"). Assim, após a primeira vez em que forem executadas, os valores de retorno não serão mais utilizados para as demais execuções. Ou seja, em vez de utilizar `mockImplementation`, `mockResolvedValue` ou `mockRejectedValue`, você pode utilizar `mockImplementationOnce`, `mockResolvedValueOnce` e `mockRejectedValueOnce` caso não queira manter o valor de retorno para as próximas execuções.

O que precisamos fazer é garantir que a função `loadDatabase` e `saveDatabase` foram chamadas corretamente (uma única vez cada) e com seus respectivos valores, caso algum parâmetro seja fornecido.

Fazemos isso através da asserção `toHaveBeenCalled`, que verifica se uma função foi chamada. Podemos utilizar também `toHaveBeenCalledTimes` para garantir que ela foi executada um número determinado de vezes e `toHaveBeenCalledWith` para verificar os

parâmetros que foram fornecidos para a função. Vamos aproveitar e trocar nosso `mockResolvedValue` para `mockResolvedValueOnce`.

```
it('Cria usuário corretamente', async () => {
  file.loadDatabase.mockResolvedValueOnce([]);
  const user = await createUser(usuario);

  // novas asserções verificando se as funções foram chamadas 1 vez
  expect(file.loadDatabase).toHaveBeenCalledTimes(1);
  expect(file.saveDatabase).toHaveBeenCalledTimes(1);
});
```

Se quiser testar, mude qualquer um dos valores para `2`, por exemplo. Você verá que nosso teste quebrará, já que as funções `loadDatabase` e `saveDatabase` foram chamadas uma vez cada.

Agora, vamos fazer uma asserção para verificar que a função `saveDatabase` foi chamada com um array contendo o novo usuário. Vamos utilizar `toHaveBeenCalledWith` para isso:

```
it('Cria usuário corretamente', async () => {
  file.loadDatabase.mockResolvedValueOnce([]);
  const user = await createUser(usuario);

  expect(file.loadDatabase).toHaveBeenCalledTimes(1);
  expect(file.saveDatabase).toHaveBeenCalledTimes(1);
  expect(file.saveDatabase).toHaveBeenCalledWith([user]);
});
```

Perfeito! Podemos realizar uma asserção para verificar se o novo usuário possui todos os campos da variável `usuario`, além de conter um `uid` do tipo `string`. Além disso, como não informamos nenhuma `role`, também devemos nos certificar de que a `role` padrão é `USER`.

```
it('Cria usuário corretamente', async () => {
  file.loadDatabase.mockResolvedValueOnce([]);
  const user = await createUser(usuario);

  expect(file.loadDatabase).toHaveBeenCalledTimes(1);
  expect(file.saveDatabase).toHaveBeenCalledTimes(1);
```

```
  expect(file.saveDatabase).toHaveBeenCalledWith([user]);
  expect(user).toEqual({
    ...usuario,
    uid: expect.any(String),
    role: 'USER'
  });
});
```

Caso não queira manipular o valor da `role` como uma string diretamente, você pode importá-la da pasta `constants` e verificar se a role é igual ao valor de `ROLE.USER` , da seguinte maneira:

```
import { createUser } from '../../../src/database/user/create.js';
import * as file from '../../../src/database/file.js';
// importamos as ROLES
import ROLES from '../../../src/constants/roles.js'

jest.mock('../../../src/database/file.js');
jest.mock('../../../src/database/path.js');

const usuario = {
  email: 'qualquer@email.com',
  password: 'senha1234',
  userName: 'usuarioQualquer',
  name: 'Usuario',
  lastName: 'Qualquer'
};

it('Cria usuário corretamente', async () => {
  file.loadDatabase.mockResolvedValueOnce([]);
  const user = await createUser(usuario);

  expect(file.loadDatabase).toHaveBeenCalledTimes(1);
  expect(file.saveDatabase).toHaveBeenCalledTimes(1);
  expect(file.saveDatabase).toHaveBeenCalledWith([user]);
  expect(user).toEqual({
    ...usuario,
    uid: expect.any(String),
    role: ROLES.USER // modificamos o valor para ROLES.USER
  });
});
```

Testando criação de usuários do tipo ADMIN

Perfeito! Nosso teste passa! Agora só precisamos nos assegurar de que, ao informar uma `role` diferente, ela será utilizada no novo registro. Podemos já aproveitar que importamos as `ROLES` das constantes e duplicar nosso teste e fazer as devidas alterações. Além de informar a variável usuário, precisamos informar a `ROLE` de `admin` também:

```
it('Cria usuário corretamente', async () => {
  file.loadDatabase.mockResolvedValueOnce([]);
  const user = await createUser(usuario);

  expect(file.loadDatabase).toHaveBeenCalledTimes(1);
  expect(file.saveDatabase).toHaveBeenCalledTimes(1);
  expect(file.saveDatabase).toHaveBeenCalledWith([user]);
  expect(user).toEqual({
    ...usuario,
    uid: expect.any(String),
    role: ROLES.USER
  });
});

// novo teste
it('Cria usuário corretamente com role ADMIN', async () => {
  file.loadDatabase.mockResolvedValueOnce([]);
  const user = await createUser({ ...usuario, role: ROLES.ADMIN }); //
passamos usuário e a role

  expect(file.loadDatabase).toHaveBeenCalledTimes(1);
  expect(file.saveDatabase).toHaveBeenCalledTimes(1);
  expect(file.saveDatabase).toHaveBeenCalledWith([user]);
  expect(user).toEqual({
    ...usuario,
    uid: expect.any(String),
    role: ROLES.ADMIN // verificamos se agora o usuário possui a role nova
  });
});
```

Não podemos esquecer também de limpar as chamadas dos nossos mocks ao final de cada teste. Atualmente, nosso teste apresentaria um erro, pois, na segunda execução, nossas funções teriam sido chamadas duas vezes já que não realizamos essa limpeza. Vamos fazer isso com o hook `afterEach` como vimos anteriormente:

```
// inserimos o hook antes de nossos testes
afterEach(() => {
  // usando jest.clearAllMocks para facilitar
  // ao invés de limpar cada uma das duas funções
  jest.clearAllMocks();
});
```

Nossos testes já funcionam de forma coerente, mas ainda há dois ajustes que podemos fazer:

- Como nosso `mockResolvedValueOnce` da função `loadDatabase` é sempre igual, podemos colocá-la em um hook `beforeEach` também, para evitar ficar alterando o valor desse mock em todos os testes.
- Ao final de tudo, no hook `afterAll`, podemos restaurar todos os mocks com o `restoreAllMocks`.

Com isso, teremos os três hooks a seguir antes de nossos testes:

```
// aplica o valor que loadDatabase deverá retornar antes de cada teste
beforeEach(() => {
  file.loadDatabase.mockResolvedValueOnce([]);
});

afterEach(() => {
  jest.clearAllMocks();
});

// restaura todos os mocks após todos os testes
afterAll(() => {
  jest.restoreAllMocks();
});
```

E pronto, nosso teste já está bem bacana. Se quiser, pode atualizar o `coverage` com o comando `npm run test:coverage` e ver como está nossa cobertura.

Caso o `coverage` aponte novos arquivos que não estão cobertos por testes, não se assuste. Como nossas funções estão ligadas através de `imports/exports` é normal que alguns outros arquivos sejam mapeados.

6.2 Um pouco mais de assincronia: garantindo cenários de falha

As funções de leitura de usuário, `getUserByUid` e `getUserByUsernameAndPassword`, que estão no arquivo `src/database/user/read.js`, disparam algumas mensagens de erro caso um usuário não seja encontrado.

A função `getUserByUid` é utilizada para tentar encontrar um usuário qualquer a partir de seu `uid` e, caso não encontre, dispara a mensagem de erro `Não existe usuário com uid informado.`. Já a função `getUserByUsernameAndPassword` retorna a mensagem `Credenciais incorretas ou usuário inexistente.` se não encontrar. Em caso de sucesso, ambas as funções retornam o usuário encontrado.

Vamos começar testando a função `getUserByUid`. Dentro da pasta `__tests__/database`, crie o arquivo `user.test.js`. Já vamos importar a função que vamos testar também e, como vimos anteriormente, já vamos realizar o mock dos arquivos `file` e `path` para que possamos testar corretamente o que queremos:

```
// arquivo __tests__/database/user/read.test.js
import { getUserByUid } from '../../../src/database/user/read.js';

jest.mock('../../../src/database/path.js');
jest.mock('../../../src/database/file.js');
```

Para criar nosso teste, também precisaremos importar a função `loadDatabase` do arquivo `src/database/file.js` e "mockar" seu resultado para algum valor que queremos. Dessa forma, simularemos o arquivo `database.json` com algum usuário para que possamos verificar os cenários de sucesso (caso algum usuário seja encontrado) e o cenário de falha (caso não encontre nenhum usuário).

Então vamos importar essa função também, visto que já realizamos o mock dela:

```
import { getUserByUid } from '../../../src/database/user/read.js';
// importamos loadDatabase
import { loadDatabase } from '../../../src/database/file.js';

jest.mock('../../../src/database/path.js');
jest.mock('../../../src/database/file.js');
```

E simular um retorno da função `loadDatabase` com somente um usuário:

```
import { getUserByUid } from '../../../src/database/user/read.js';
import { loadDatabase } from '../../../src/database/file.js';

jest.mock('../../../src/database/path.js');
jest.mock('../../../src/database/file.js');

// simulamos array de usuários contendo apenas um
loadDatabase.mockResolvedValue([
  {
    uid: 'abc-1234',
    userName: 'nomeDeUsuario',
    name: 'nome',
    lastName: 'DeUsuario',
    email: 'email.nome@usuario.com',
    password: 'senhasupersecreta',
    role: 'USER'
  }
]);
```

Vamos começar a criar nosso primeiro cenário de teste. Como usamos Promises, não esqueça de colocar `async` na assinatura para facilitar nossa leitura e poder utilizar `await` (poderia ser feito sem, mas nos ajuda a manter um padrão):

```
import { getUserByUid } from '../../../src/database/user/read.js';
import { loadDatabase } from '../../../src/database/file.js';

jest.mock('../../../src/database/path.js');
jest.mock('../../../src/database/file.js');

loadDatabase.mockResolvedValue([
  {
    uid: 'abc-1234',
    userName: 'nomeDeUsuario',
    name: 'nome',
    lastName: 'DeUsuario',
    email: 'email.nome@usuario.com',
    password: 'senhasupersecreta',
    role: 'USER'
  }
]);

// iniciamos caso de teste
it('Encontra usuário quando encontra seu UID', async() => {
});
```

Para que possamos reutilizar os dados do usuário que "mockamos", vamos extraí-lo para uma variável. Podemos aproveitar e utilizar essa variável no array retornado que informamos à função `mockResolvedValue`:

```
import { getUserByUid } from '../../../src/database/user/read.js';
import { loadDatabase } from '../../../src/database/file.js';

jest.mock('../../../src/database/path.js');
jest.mock('../../../src/database/file.js');

// extraímos dados para uma variável mockUsuario
const mockUsuario = {
```

```
  uid: 'abc-1234',
  userName: 'nomeDeUsuario',
  name: 'nome',
  lastName: 'DeUsuario',
  email: 'email.nome@usuario.com',
  password: 'senhasupersecreta',
  role: 'USER'
};

// utilizamos variável no array
loadDatabase.mockResolvedValue([mockUsuario]);

it('Encontra usuário quando encontra seu UID', async() => {
});
```

Para o cenário de sucesso será fácil. Tudo o que precisamos fazer é executar a função `getUserByUid` informando o `uid` do usuário falso que informamos como retorno do nosso mock:

```
it('Encontra usuário quando encontra seu UID', async() => {
  const usuario = await getUserByUid('abc-1234');
  expect(usuario.userName).toEqual('nomeDeUsuario');
});
```

Para que não precisemos ficar verificando cada um dos dados do retorno, podemos aproveitar a variável `mockUsuario`, que criamos:

```
it('Encontra usuário quando encontra seu UID', async() => {
  const usuario = await getUserByUid('abc-1234');
  expect(usuario).toEqual(mockUsuario);
});
```

Agora vamos iniciar nosso cenário de erro:

```
it('Encontra usuário quando encontra seu UID', async() => {
  const usuario = await getUserByUid('abc-1234');
  expect(usuario).toEqual(mockUsuario);
});

// inicia cenário de erro
```

```
it('Dispara um erro caso usuário não seja encontrado', async() => {
});
```

Para testar esse cenário, precisaremos criar um bloco `try/catch` dentro do nosso teste e colocar a execução da função `getUserByUid` com um `uid` inválido dentro do bloco `try` :

```
it('Dispara um erro caso usuário não seja encontrado', async() => {
  // criamos bloco try/catch
  try {
    // executamos getUserByUid com um uid invalido
    await getUserByUid('uid-nao-existente');
  } catch (err) {
  }
});
```

A seguir, o que devemos fazer é colocar uma asserção dentro do bloco `catch` com o que quisermos. Podemos, por exemplo, verificar se o campo `message` de `err` é igual ao erro `Não existe usuário com uid informado.` :

```
it('Dispara um erro caso usuário não seja encontrado', async() => {
  try {
    await getUserByUid('uid-nao-existente');
  } catch (err) {
    // adicionamos asserção usando a mensagem de erro
    expect(err.message).toEqual('Não existe usuário com uid informado.');
  }
});
```

Claro que poderíamos criar algum utilitário para gerar os erros de nossa aplicação de forma mais prática, isso também nos ajudaria a não ficar lidando com as strings de erro manualmente em nossos testes. Deixo mais esse desafio para você: criar esse utilitário, testá-lo e aplicá-lo nesses testes que disparam uma mensagem de erro.

6.3 Lembrete importante quando trabalhamos com testes assíncronos

Em alguns cenários, ao testar código assíncrono, é muito fácil gerar um teste com falso-positivo. Lembra quando fizemos um teste com somente uma Promise qualquer e o teste passava? Vamos criar um cenário (nesse mesmo arquivo) para entendermos melhor:

```
// novo teste
it('Deve conter pelo menos 1 asserção', async() => {
  await Promise.resolve(1);
});
```

Ao salvar o arquivo, você verá que esse teste, mesmo sem conter nenhuma asserção, passará. Isso acontece porque, ao utilizar `async/await`, nossa função é interrompida e retornada com a Promise no estado em que ela se encontra (bem parecido com `generators`).

Vamos pensar em um cenário de erro, como no nosso segundo teste. Se um dia alguma pessoa modificar a lógica da função `getUserByUid` e ela não disparar mais um erro, podemos ter um teste que passa, mas que não estará com suas asserções sendo executadas.

Para contornar esses cenários, o Jest nos provê uma função chamada `expect.assetions`. Ela recebe um número como parâmetro e devemos informar a ela a quantidade de asserções que um teste terá até o final de sua execução.

No caso do nosso novo teste, por exemplo, podemos indicar que ele teria somente uma asserção, da seguinte forma:

```
it('Deve conter pelo menos 1 asserção', async() => {
  // inserimos expect.assertions indicando somente 1 asserção
  expect.assertions(1);
  await Promise.resolve(1);
});
```

Agora sim, se salvarmos o arquivo, teremos o seguinte erro:

```
Expected one assertion to be called but received zero assertion calls.
```

Então, podemos adotar isso como uma prática bacana de se fazer, principalmente ao lidar com testes assíncronos. Vamos aproveitar e aplicar isso nos testes que fizemos anteriormente:

```
it('Encontra usuário quando encontra seu UID', async() => {
  expect.assertions(1); // inserimos aqui
  const usuario = await getUserByUid('abc-1234');
  expect(usuario).toEqual(mockUsuario);
});

it('Dispara um erro caso usuário não seja encontrado', async() => {
  expect.assertions(1); // inserimos aqui
  try {
    await getUserByUid('uid-nao-existente');
  } catch (err) {
    expect(err.message).toEqual('Não existe usuário com uid informado.');
  }
});
```

E ajustar os testes em `__tests__/database/user/create.test.js` também:

```
it('Cria usuário corretamente', async () => {
  expect.assertions(4);
  // asserções omitidas
});

it('Cria usuário corretamente com role ADMIN', async () => {
  expect.assertions(4);
  // asserções omitidas
});
```

Assim garantimos que todos os nossos códigos assíncronos utilizam essa função.

O que acha de dar uma olhada em algumas configurações extras?

Podemos customizar várias coisas através das configurações do Jest, uma delas é o mapeamento dos arquivos que importamos. No próximo capítulo, vamos realizar essa melhoria e aproveitar para testarmos mais uma camada da aplicação CLI, os middlewares.

CAPÍTULO 7
Ajustando configurações e testando middlewares

Podemos realizar alguns ajustes de configuração para que fique mais fácil de importar os arquivos em nossos testes.

É sempre interessante se aprofundar nas ferramentas que você utiliza no dia a dia e podemos compreender melhor como o Jest funciona ao lidar com suas configurações e customizar nosso projeto. Nesse momento, vamos realizar um ajuste para poder tornar a escrita e a execução dos nossos testes mais amigáveis.

7.1 Facilitando acesso aos arquivos do projeto

Se você notou bem, em todos os arquivos de teste, estamos voltando manualmente várias pastas para conseguirmos importar os módulos que queremos. Por exemplo:

```
// voltamos 3 diretórios para encontrar o arquivo que queremos
import { getUserByUid } from '../../../src/database/user/read.js';
```

E essa não é uma tarefa muito bacana de ficarmos repetindo. Vamos utilizar aquele arquivo de configuração do Jest para mapear alguns módulos para nós. Ao final desse processo, teremos transformado esses imports em algo mais ou menos assim:

```
import logger from '../../src/utils/logger.js';
```

Para fazer isso, vamos criar o arquivo `jest.config.js` na raiz do nosso projeto:

```
// arquivo jest.config.js
export default {}
```

Por enquanto, ele somente exporta como padrão um objeto vazio. Para que possamos indicar o caminho dos módulos que queremos, utilizaremos a chave `moduleNameMapper`. Essa chave receberá um objeto dentro do qual vamos configurar alguns apelidos (alias) para algumas pastas de nosso projeto e informar o caminho real que o Jest usará para importar o arquivo para o teste, da seguinte forma:

```
export default {
  moduleNameMapper: {
    '^utils/(.*)$': '<rootDir>/src/utils/$1',
  }
}
```

Vamos entender tudo o que aconteceu. Dentro do objeto `moduleNameMapper`, foi criada uma chave utilizando a expressão regular `^utils/(.*)$`, que indicará ao Jest esse novo caminho, e foi atribuído o valor de `'<rootDir>/src/utils/$1'` para essa chave.

Isso indica que, ao ter algum import dentro de um teste como `utils/qualquerArquivo.js`, o Jest deverá procurar por `qualquerArquivo.js` a partir da raiz do projeto (e, por isso, `<rootDir>`) dentro da pasta `src/utils/`.

Com isso, nosso import que antes era desta forma:

```
import logger from '../../src/utils/logger.js';
```

Ficará assim:

```
import logger from 'utils/logger.js';
```

Bem mais prático, não acha?

Com isso, podemos configurar os diretórios principais da nossa aplicação para essa estrutura e fazer as mudanças de nossos imports.

7.2 Entendendo cadeia de responsabilidades ao testar middlewares

No projeto, existe uma pasta `middlewares` com arquivos e funções responsáveis por executar algumas determinadas ações antes das operações de CRUD (leitura/criação/atualização/remoção de usuário).

Antes de vermos como eles funcionam, vamos tentar entender como implementaríamos algumas validações dentro de nossas operações. Por exemplo, para executar a operação de criar um usuário, precisamos realizar algumas etapas antes:

- Verificar se o usuário que está utilizando a CLI possui privilégios de admin;
- Verificar se todos os campos corretos foram enviados no campo `data` ;
- Criar o usuário se todas as verificações anteriores estiverem corretas.

À primeira vista, pode ser tentador colocar todas essas validações diretamente na nossa operação de criação de usuário com alguns blocos de `if/else` , no entanto muitas dessas verificações são utilizadas para as demais operações também. Ou seja, em todas as operações teríamos que ficar manualmente inserindo várias regras de validação. Quanto maiores essas regras, maiores nossas operações e validações se tornariam. Por exemplo, vamos esboçar um trecho de código:

```
const criaUsuario = ({ dados, usuario }) => {
  if (!dadosValidos(dados)) {
    // faz alguma operação caso os dados não estejam válidos
  }
  if (!usuarioEhAdmin(usuario)) {
    // faz alguma operação caso o usuário não possa executar operação
  }
}
```

Conforme nossa aplicação fosse crescendo, seria um pouco desgastante ficar dando manutenção e testando. Sem contar que, com todas essas modificações, nossas operações não estariam responsáveis somente por realizar alguma operação de fato, mas elas também teriam internamente muitas regras que não seriam necessárias. Além de tudo, nossa função ficou acoplada a diversas validações, o que não é necessário. Se fôssemos imaginar um fluxograma do nosso código com o exemplo acima, teríamos algo assim:

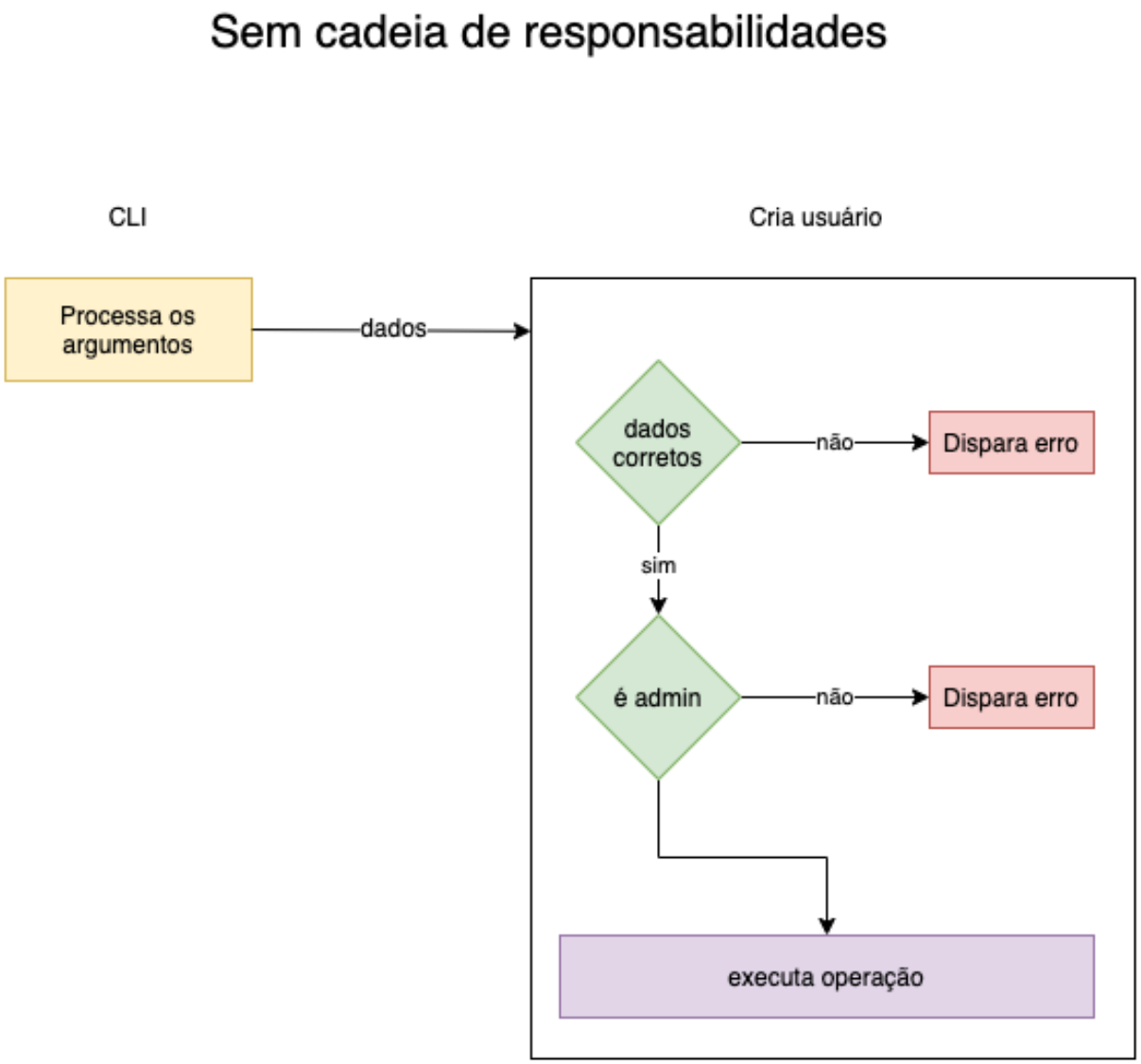


Figura 7.1: Validações sequenciais diretamente na função.

Em um cenário mais organizado, as operações seriam executadas somente se todas as validações anteriores já tivessem ocorrido e estivesse tudo certo para prosseguirmos com a operação em si. Dessa forma, a função `criaUsuario` (por exemplo) nem seria executada caso todos os requisitos necessários para que ela funcionasse não fossem cumpridos.

Para chegar a esse cenário onde organizamos nossas responsabilidades de forma mais coerente e também desacoplamos nossas validações de nossa operação, podemos utilizar um padrão de projeto (ou *design pattern*) chamado **cadeia de responsabilidade** (ou *chain of responsibility*).

Esse padrão, basicamente, indica uma estrutura onde podemos criar uma cadeia de funções que serão responsáveis por continuar ou não com a execução das funções seguintes. Nesse cenário, teríamos a função que valida os dados e a função que valida se o usuário é administrador de forma separadas. Se qualquer uma delas encontrar alguma anomalia, como o usuário não ter permissão para executar a ação ou não ter fornecido os dados corretos, essa mesma função pode decidir interromper a execução das demais. Se tudo estiver certo, as funções seguintes podem prosseguir.

Aplicando esse padrão, nosso fluxo fica algo mais ou menos assim:

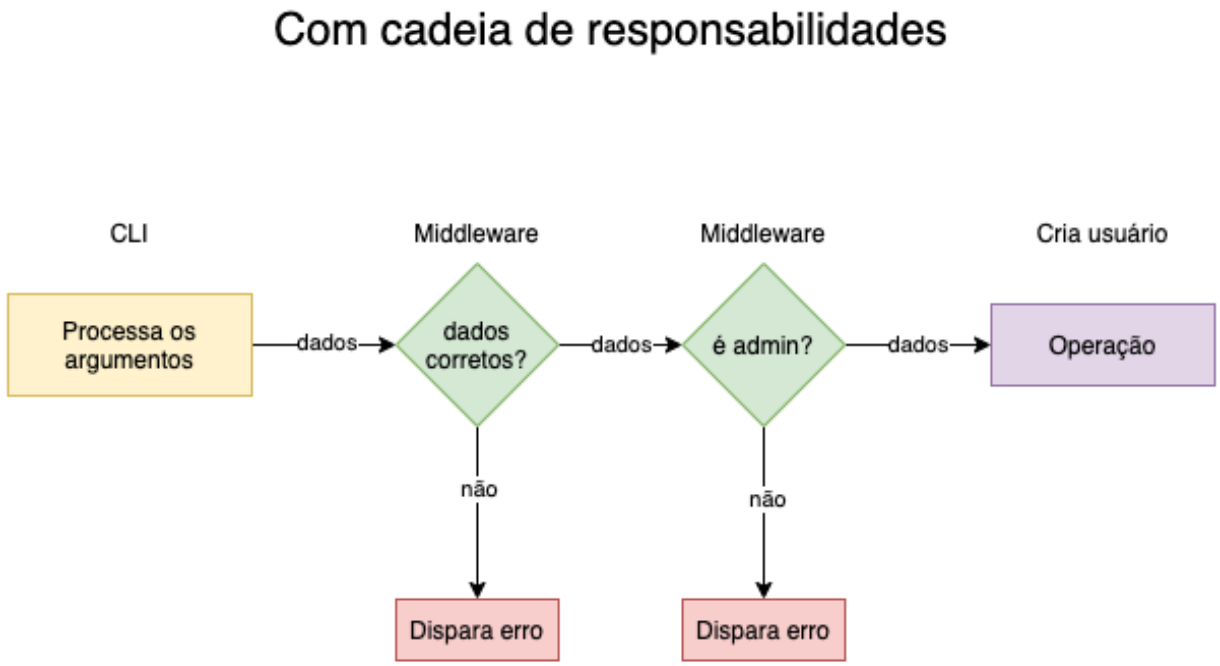


Figura 7.2: Validações com cadeia de responsabilidades.

Nesse padrão, cada uma dessas funções intermediárias recebe um nome que talvez você já conheça: *middleware*.

Um middleware é basicamente uma função intermediária (por isso recebe esse nome) qualquer que é executada antes de uma ou mais funções em uma cadeia de execução. Essas funções têm um certo "poder" de decidir como será o restante do fluxo de execução: podem interromper (lançando um erro, por exemplo) e podem, além de deixar tudo prosseguir como o esperado, adicionar informações nos parâmetros das funções seguintes.

Você pode estar se perguntando: **por que aplicar toda essa complexidade**? E a resposta é simples: justamente para que, conforme uma aplicação cresça, sua manutenção e inserção de novos requisitos sejam menos complexas. Sem contar que reutilizar esses middlewares é algo muito mais simples e organizado do que ficar manipulando blocos de `if/else` manualmente.

À primeira vista, pode parecer um padrão complicado, mas é bem simples de se colocar em prática. Esse fluxo que vimos é exatamente a composição dos middlewares da operação de criação de usuário. Olhando no arquivo `src/operations/create.js`, você consegue ver que, em suas últimas linhas, existe um `export` que compõe todos esses middlewares nesta exata ordem que comentei:

```
// arquivo src/operations/create.js
export default applyMiddlewares(
  isAdminMiddleware,
  validateDataMiddleware(['email', 'password', 'userName', 'name',
'lastName']),
  create
);
```

Se qualquer uma dessas funções de middleware não for "satisfeita" e os dados não forem válidos, elas lançam uma exceção e interrompem o funcionamento da cadeia. Em outras palavras, a função `create` (assim como as demais operações) só é executada se estiver tudo correto para o seu funcionamento.

A função `applyMiddlewares` é bem simples. Ela está no arquivo `src/middlewares/index.js` e, basicamente, faz um `reduce` de todas as funções fornecidas e passa todos os argumentos para a frente. Vale a pena dar uma olhada nela (claro que é uma implementação bem singela).

Começando os testes dos middlewares

Após essa conversa sobre middlewares e cadeia de responsabilidade, vamos então criar alguns testes. Vamos começar testando a função `applyMiddlewares`. Portanto, vamos criar essa mesma estrutura de pasta no nosso diretório `__tests__`, criar um arquivo `index.test.js` e importar essa função:

```
// arquivo __tests__/middlewares/index.test.js
import applyMiddlewares from 'middlewares/index.js';
```

Note que eu já estou utilizando os caminhos no `import` da forma mais curta, já com a sua devida configuração no `moduleNameMapper` no arquivo `jest.config.js`.

Testar essa função será bem simples, tudo o que precisaremos fazer é criar duas funções de mock (`jest.fn`). Essas duas funções devem receber e retornar um mesmo valor em suas implementações. Vamos criar um caso de teste com elas:

```
// caso de teste criado
it('Deve retornar uma nova função que chama os demais middlewares ao ser
executada', () => {
  // duas funções que retornam os valores que recebem
  const mid1 = jest.fn(data => data);
  const mid2 = jest.fn(data => data);
});
```

Precisamos executar a função `applyMiddleware` passando essas duas variáveis que criamos:

```
it('Deve retornar uma nova função que chama os demais middlewares ao ser
executada', () => {
```

```
  const mid1 = jest.fn(data => data);
  const mid2 = jest.fn(data => data);

  // criamos nova variável após aplicar os middlewares
  const middlewaresAplicados = applyMiddlewares(mid1, mid2);
});
```

Isso nos retornará uma função, que deverá receber um argumento e repassá-lo para os middlewares que aplicamos. Neste momento, podemos nos certificar de que, por enquanto, `middlewaresAplicados` é uma função e que nossas variáveis `mid1` e `mid2` não foram chamadas:

```
it('Deve retornar uma nova função que chama os demais middlewares ao ser executada', () => {
  const mid1 = jest.fn(data => data);
  const mid2 = jest.fn(data => data);

  const middlewaresAplicados = applyMiddlewares(mid1, mid2);

  // verificamos se middlewaresAplicados é uma função
  expect(middlewaresAplicados).toEqual(expect.any(Function));
  // verificamos que mid1 e mid2 não foram chamadas
  expect(mid1).not.toHaveBeenCalled();
  expect(mid2).not.toHaveBeenCalled();
});
```

Agora, precisamos executar a função `middlewaresAplicados` passando qualquer parâmetro para ela. Podemos então criar uma variável contendo qualquer valor para simular esse parâmetro e depois executar a função com esse mesmo valor:

```
it('Deve retornar uma nova função que chama os demais middlewares ao ser executada', () => {
  const mid1 = jest.fn(data => data);
  const mid2 = jest.fn(data => data);

  const middlewaresAplicados = applyMiddlewares(mid1, mid2);

  expect(middlewaresAplicados).toEqual(expect.any(Function));
```

```
  expect(mid1).not.toHaveBeenCalled();
  expect(mid2).not.toHaveBeenCalled();

  const argumentos = 'dados'; // criamos variável
  middlewaresAplicados(argumentos); // executamos função
});
```

A seguir, precisamos fazer mais algumas asserções, verificando que `mid1` e `mid2` foram chamadas uma única vez cada, também recebendo o valor de `argumentos` como parâmetro:

```
it('Deve retornar uma nova função que chama os demais middlewares ao ser executada', () => {
  const mid1 = jest.fn(data => data);
  const mid2 = jest.fn(data => data);

  const middlewaresAplicados = applyMiddlewares(mid1, mid2);

  expect(middlewaresAplicados).toEqual(expect.any(Function));
  expect(mid1).not.toHaveBeenCalled();
  expect(mid2).not.toHaveBeenCalled();

  const argumentos = 'dados';
  middlewaresAplicados(argumentos);

  // asserção que verifica se mid1 e mid2 foram chamadas
  // uma única vez e receberam o valor de argumentos como parâmetro
  expect(mid1).toHaveBeenCalledTimes(1);
  expect(mid1).toHaveBeenCalledWith(argumentos);
  expect(mid2).toHaveBeenCalledTimes(1);
  expect(mid2).toHaveBeenCalledWith(argumentos);
});
```

Para finalizar nossa primeira etapa, chegou a hora de testar a função `isAdminMiddleware`, localizada no arquivo `src/middlewares/user.js`. Portanto, vamos criar um arquivo `user.test.js` dentro da pasta `__tests__/middlewares/` e importar essa função. Vamos aproveitar e importar as nossas `ROLES` também:

```
// arquivo __tests__/middlewares/user.test.js
import { isAdminMiddleware } from 'middlewares/user.js'
```

```
import ROLES from 'constants/roles.js'
```

O que esse middleware faz, basicamente, é receber um objeto com os dados de um usuário dentro. Caso esse usuário possua a `role` de `ADMIN`, nada acontecerá e o middleware retornará esse mesmo usuário que recebeu para que as próximas funções na cadeia possam utilizar esses valores também. Entretanto, caso o usuário não possua a `role` correta, ele vai disparar um erro com a mensagem `Você não possui permissão para executar essa operação.`, portanto, são esses dois comportamentos que precisamos testar!

Vamos criar uma variável contendo alguns dados de usuário (sem o valor de `role`) e também iniciar nosso cenário de teste:

```
import { isAdminMiddleware } from 'middlewares/user.js'
import ROLES from 'constants/roles.js'

// criamos o mock
const mockUsuario = {
  uid: 'abc-1234',
  userName: 'nomeDeUsuario',
  name: 'nome',
  lastName: 'DeUsuario',
  email: 'email.nome@usuario.com',
  password: 'senhasupersecreta',
};

it('Deve retornar os dados do usuário caso a role seja ADMIN', () => {
  // vamos iniciar o teste
});
```

Tudo o que precisaremos fazer é chamar a função `isAdminMiddleware` e informar um objeto contendo nosso `mockUsuario` junto da `role` com o valor `ADMIN`. Vamos também fazer uma asserção e garantir que os dados retornados são iguais aos que foram passados como parâmetro:

```
it('Deve retornar os dados do usuário caso a role seja ADMIN', () => {
  // criamos um objeto contendo todos os dados de mockUsuario
  // dentro do objeto user
```

```
  // e também inserindo a role admin
  const mockAdmin = {
    user: {
      ...mockUsuario,
      role: ROLES.ADMIN,
    }
  };


  // verificamos se o retorno do middleware é igual ao valor que
informamos
  const retorno = isAdminMiddleware(mockAdmin);
  expect(retorno).toEqual(mockAdmin);
});
```

Perfeito! Nosso teste já está passando e validando o "caminho feliz" do nosso middleware. Chegou a hora de validar o caminho não tão feliz assim. Com isso em mente, vamos criar mais um caso de teste e criar uma variável de usuário nova, como fizemos no teste anterior:

```
it('Deve retornar os dados do usuário caso a role seja ADMIN', () => {
  const mockAdmin = {
    user: {
      ...mockUsuario,
      role: ROLES.ADMIN,
    }
  };

  const retorno = isAdminMiddleware(mockAdmin);
  expect(retorno).toEqual(mockAdmin);
});

// novo teste
it('Deve disparar um erro caso o usuário não seja ADMIN', () => {
  // nova variável com role de usuario
  const mockUser = {
    user: {
      ...mockUsuario,
      role: ROLES.USER,
```

```
    }
  };

  // atualizamos o expect informando mockUser como parâmetro para a função
  const retorno = isAdminMiddleware(mockUser);
  expect(retorno).toEqual(mockAdmin);
});
```

No entanto, precisaremos garantir que nossa função dispara uma mensagem de erro. Para fazer isso, trocaremos o `toEqual` para `toThrow`:

```
// código acima omitido
it('Deve disparar um erro caso o usuário não seja ADMIN', () => {
  const mockUser = {
    user: {
      ...mockUsuario,
      role: ROLES.USER,
    }
  };

  const retorno = isAdminMiddleware(mockUser);
  expect(retorno).toThrow('Você não possui permissão para executar essa
operação.');
});
```

Porém, ainda falta mais um detalhe para que nossa asserção esteja válida: quando estamos trabalhando com casos que podem resultar em um erro, precisamos passar para o `expect` uma função. Isso acontece pois a execução do nosso código é interrompida ao disparar um erro em qualquer momento dentro do teste. Fazendo isso, deixamos que o próprio Jest execute essa função e lide com esse erro internamente, permitindo-nos fazer asserções com ele.

O que precisaremos fazer é passar uma função que executará `isAdminMiddleware` e podemos fazer isso da forma mais simples, como:

```
it('Deve disparar um erro caso o usuário não seja ADMIN', () => {
  const mockUser = {
```

```
    user: {
      ...mockUsuario,
      role: ROLES.USER,
    }
  };

  // agora o retorno é uma função que será executada no expect
  const retorno = () => isAdminMiddleware(mockUser);
  expect(retorno).toThrow('Você não possui permissão para executar essa
operação.');
});
```

E tudo deu certo, nosso teste está completamente válido!

Mais um passo concluído. Próxima parada: aplicações back-end!

Demos mais um passo adiante na nossa caminhada ao lado dos testes em JavaScript. Tenho certeza de que sua confiança no código que você escreve está mais fortalecida a cada novo cenário de teste.

Com tudo o que aprendeu até aqui, você já consegue desenvolver os testes de todos os arquivos restantes da nossa aplicação `CLI`. Então, para finalizar esta segunda parte do nosso livro, deixo mais esse desafio para que você possa praticar um pouco mais, lembrando que todos os exemplos com testes completos estão no repositório que usamos para baixar a aplicação, https://javascriptassertivo.com.br/, e lá você consegue dar uma olhada em tudo o que foi desenvolvido caso precise.

Bons testes e vamos para a próxima etapa!

Parte 3: Testando aplicações back-end

Chegou a hora de testar aplicações back-end e não ficar mais somente no terminal.

Com uma aplicação em Node com Express que consome as funcionalidades da CLI criada anteriormente, realizaremos todos os testes de unidade e integração. Para finalizar, também aplicaremos testes de carga em nossa API para colhermos algumas métricas de como ela se comporta com determinadas quantidades de acesso.

Capítulo 8
Testes unitários com Node e Express

A aplicação que vamos testar nesta parte do livro também está no mesmo repositório das demais: https://javascriptassertivo.com.br/. Acessando a pasta `projetos/03-testando-aplicacoes-backend` você consegue encontrar o código que usaremos para trabalhar nossos testes. Essa aplicação se baseia em Node e expõe uma API tendo Express como framework para a criação de suas rotas, além de aplicar o projeto criado anteriormente (a CLI) em formato de API, para que requisições HTTP possam ser feitas.

Assim como o projeto anterior, você pode apagar a pasta `__tests__` e `utils` para que possamos escrever os testes juntos. Aproveite e remova o arquivo `jest.config.js` e `jest.setup.js` também, pois vamos criá-los daqui a pouco.

8.1 Estrutura do projeto, instalação e configuração

Como no projeto anterior, existe um arquivo `README.md` na pasta raiz que possui mais alguns detalhes sobre a construção do projeto e sua estrutura. Vamos dar uma olhada nas pastas principais:

- `http` : onde você pode interagir com as rotas da API. Com o arquivo `api.http` , você consegue disparar as requisições usando extensões como REST Client (disponível para download no VSCode https://marketplace.visualstudio.com/items?itemName=humao.rest-client) ou, caso prefira, pode utilizar a collection do Postman através do arquivo `postman_collection.json` ;
- `src` : contém o código-fonte da aplicação com as seguintes subpastas e arquivos:
  - `controllers` : contém o código que vai interagir, de fato, com uma requisição ao final de uma cadeia de `middlewares` , para as rotas de `auth` , `user` e `users` ;
  - `middlewares` : seguem o mesmo fundamento que vimos na CLI;
  - `routes` : contém as especificações e criação das rotas da API, sendo `auth` , `users` e `users` ;
  - `services` : contém alguns métodos para listagem de usuários que não foram criados na CLI anteriormente, mas são interessantes para a aplicação. Também atua como uma camada intermediária, acessando os dados da CLI e aplicando os filtros caso necessário;
  - `index.js` : exporta a função principal que inicia o servidor e as rotas.

Esse projeto foi desenvolvido utilizando como base todo o código da CLI criada anteriormente. Apenas foi realizada uma "casca" de uma aplicação utilizando Express para expor uma API que permite a consulta e a manipulação dos dados.

Note que, no arquivo `package.json` , existe a seguinte linha:

```
{
    "dependencies": {
        "@jsassertivo/cli": "file://../02-aplicando-testes-unitarios-em-uma-cli/"
    }
}
```

O que essa linha faz é instalar o código da aplicação do projeto anterior como uma dependência dessa nova aplicação, atribuindo o nome de `jsassertivo`. A qualquer momento que você visualizar algum `import`, como:

```
import { getUserByUid, getUserByUsernameAndPassword } from '@jsassertivo/cli/src/database/user/read.js';
```

Tenha em mente que o caminho `@jsassertivo/cli/*` é o do projeto da CLI anterior. Com isso, temos uma certa integração entre nossos projetos.

Ao utilizarmos a API feita em Express, é possível consultar e manipular o arquivo `database.json`, que simula o nosso banco de dados perfeitamente. Podemos também reutilizar a grande maioria das funções da CLI que foi desenvolvida anteriormente, desde suas operações até seus middlewares.

Para ambientar-se um pouco, instale o projeto através do comando `npm i` e execute a aplicação usando `npm start` ou `npm run start:debug`. Você pode disparar as requisições com o arquivo `http/api.http` caso tenha a extensão instalada, ou utilizar o arquivo com a coleção do Postman para ajudar nessa tarefa. Execute algumas requisições e familiarize-se com o código desenvolvido, pois aprenderemos algumas coisas bem legais a partir de agora.

8.2 Ajustando a configuração inicial

Vamos iniciar nossa configuração para que possamos realizar os testes desta etapa. Vamos criar nosso arquivo `jest.config.js` na raiz da aplicação.

Para testar essa aplicação, precisaremos realizar um mock do arquivo `database/path.js` da aplicação que contém o código da CLI, como fizemos anteriormente. No entanto, para automatizar esse trabalho, podemos indicar ao Jest um arquivo que será carregado toda vez que os testes iniciarem, justamente para realizar qualquer pré-configuração necessária.

Para fazer isso, no arquivo `jest.config.js`, basta criar a chave `setupFiles`, que será um array com esses arquivos de configuração. Vamos chamar esse arquivo de `jest.setup.js`, por exemplo:

```
// arquivo jest.config.js
export default {
  setupFiles: [
    '<rootDir>/jest.setup.js'
  ]
};
```

Assim como fizemos para cadastrar nossos `aliases` com o `moduleNameMapper`, indicamos que o arquivo deve ser carregado a partir da raiz do projeto com `<rootDir>`. Vamos criar esse arquivo `jest.setup.js` e também realizar o mock do arquivo `path.js` lá dentro:

```
// arquivo jest.setup.js
jest.mock('@jsassertivo/cli/src/database/path.js');
```

Pronto! Agora toda vez que executarmos nossos testes, o mock do arquivo será realizado.

Vamos aproveitar e voltar ao arquivo `jest.config.js` para aplicar o `moduleNameMapper` das três pastas principais, que testaremos unitariamente, `controllers`, `middlewares` e `services`:

```
export default {
  setupFiles: [
```

```
    '<rootDir>/jest.setup.js'
  ],
  // adicionamos o moduleNameMapper com os diretórios que testaremos
  moduleNameMapper: {
    '^controllers/(.*)$': '<rootDir>/src/controllers/$1',
    '^middlewares/(.*)$': '<rootDir>/src/middlewares/$1',
    '^services/(.*)$': '<rootDir>/src/services/$1'
  }
};
```

Não se preocupe, pois os arquivos da pasta `routes` servem apenas para definição das rotas da API. Poderíamos testá-los unitariamente também, mas não seria um teste que garantiria muita confiança para nós, já que esses arquivos não apresentam nenhuma lógica e apenas utilizam os métodos do Express. Garantiremos todos esses fluxos mais complexos quando chegarmos aos nossos testes de integração.

Aproveitando, como vamos testar essa aplicação de forma unitária e integrada, vamos mudar a nomenclatura dos nossos arquivos de testes para esse projeto, ok? Vamos nomear arquivos contendo os testes de unidade como `arquivo.unit.js` , e os de integração, como `arquivo.integration.js` .

8.3 Testando controller de Autenticação

Vamos iniciar nossos testes pelo controller de autenticação: o arquivo `src/controllers/auth.controller.js` . Precisamos replicar essa estrutura de pastas dentro da pasta `__tests__` , criando o arquivo `auth.controller.unit.js` . Vamos aproveitar e importar a função que esse controller exporta, a função `authenticate` :

```
// arquivo __tests__/controllers/auth.controller.unit.js
import { authenticate } from 'controllers/auth.controller';
```

A lógica desse controller é bem simples. Ele utiliza a função `findUser.usernameAndPassword` presente no arquivo `services/user/find.js` para encontrar o campo `uid` (unique-identifier) de um usuário. Caso seja encontrado, esse `uid` é utilizado como cookie. Caso não seja encontrado, é retornada uma mensagem de erro a quem fez a requisição.

Como vamos precisar verificar se esses dois módulos são utilizados, vamos aproveitar e importá-los também, já realizando seus respectivos mocks:

```
// Controller
import { authenticate } from 'controllers/auth.controller';

// Mock Logger
import logger from '@jsassertivo/cli/src/utils/logger.js';
jest.mock('@jsassertivo/cli/src/utils/logger.js');

// Mock Service
import findUser from 'services/user/find';
jest.mock('services/user/find');
```

Vamos dar uma olhada na função do controller que testaremos:

```
export const authenticate = async (req, res) => {
  try {
    const { uid } = await findUser.usernameAndPassword(req.body.username,
req.body.password);
    res.cookie('uid', uid);

    return res.json({ uid });
  } catch(err) {
    logger.error('Ocorreu um erro ao autenticar usuários', err);
    return res.status(404).json(err);
  }
}
```

Podemos ver que temos um bloco com `try/catch` onde tentamos encontrar um usuário. Essa função, assim como a grande maioria das funções relacionadas ao Express, recebe como parâmetro um

objeto com a requisição (`req`) e outro com a resposta (`res`). Os campos `username` e `password` são fornecidos a partir de um objeto presente em `req.body` , que contém o corpo da requisição. Caso a função `findUser.userNameAndPassword` retorne um objeto com `uid` , esse mesmo valor é colocado em cookie (`res.cookie`) e é inserido na resposta (`res.json`).

Caso o usuário não seja encontrado, a função `find.usernameAndPassword` retornará um erro, `logger.error` será chamado, a resposta terá um status 404 (res.status) e receberá também objeto de erro.

Testando usuário autenticado com sucesso

Vamos testar o primeiro cenário, onde o usuário é encontrado. Para isso, vamos criar nosso primeiro caso de teste e também dois objetos que simularão `req` e `res` . O objeto `req` deverá conter um campo `username` e outro `password` dentro de `body` , enquanto o objeto `res` deverá conter as funções `json` e `cookie` , que serão as funções de mock padrão do Jest. O único detalhe é que, como algumas chamadas dessas funções são encadeadas (como `res.status().json()`), precisamos lembrar de retornar o objeto `res` de cada uma delas. Como vamos utilizar Promises, vamos lembrar de colocar `async` na assinatura da função de teste também. Podemos fazer tudo isso da seguinte forma:

```
// criamos caso de teste
it('Encontra o usuário e insere seu UID em cookie', async () => {
  // criamos objeto da requisição com os campos necessários
  const req = {
    body: {
      username: 'usuario',
      password: 'senha-super-secreta'
    }
  };

  // criamos o objeto da resposta, com as funções
  // e cada função retorna o próprio objeto res
```

```
  const res = {
    json: jest.fn(() => res),
    cookie: jest.fn(() => res)
  };
});
```

Agora precisamos apenas simular o cenário onde a função `find.usernameAndPassword` retorna os dados de um usuário válido. Fazemos isso com o nosso utilitário já conhecido `mockResolvedValueOnce`:

```
it('Encontra o usuário e insere seu UID em cookie', async () => {
  const req = {
    body: {
      username: 'usuario',
      password: 'senha-super-secreta'
    }
  };

  const res = {
    json: jest.fn(() => res),
    cookie: jest.fn(() => res)
  };

  // criamos um objeto qualquer com dados de usuário e indicamos que será
  // o objeto resolvido pela função find.usernameAndPassword
  const campos = {
    uid: 'qualquer-uid',
    userName: 'username',
    password: 'password'
  };
  findUser.usernameAndPassword.mockResolvedValueOnce(campos);
});
```

Tudo o que precisamos fazer nesse momento é executar a função `authenticate` fornecendo como parâmetros as variáveis `req` e `res` e, após isso, verificar se as funções do objeto `res` foram chamadas corretamente. Podemos também verificar se a função `findUser.usernameAndPassword` foi chamada:

```
it('Encontra o usuário e insere seu UID em cookie', async () => {
  const req = {
    body: {
      username: 'usuario',
      password: 'senha-super-secreta'
    }
  };

  const res = {
    json: jest.fn(() => res),
    cookie: jest.fn(() => res)
  };

  const campos = {
    uid: 'qualquer-uid',
    userName: 'username',
    password: 'password'
  };
  findUser.usernameAndPassword.mockResolvedValueOnce(campos);

  // executamos o controller
  await authenticate(req, res);

  // verificamos se a função findUser.usernameAndPassword foi chamada
  // uma única vez e com os dados de usuario/senha
  expect(findUser.usernameAndPassword).toHaveBeenCalledTimes(1);

expect(findUser.usernameAndPassword).toHaveBeenCalledWith(req.body.usernam
e, req.body.password);

  // verificamos se res.cookie foi executada
  // uma única vez e com o uid
  expect(res.cookie).toHaveBeenCalledTimes(1);
  expect(res.cookie).toHaveBeenCalledWith('uid', uid);

  // verificamos se res.json foi executada
  // uma única vez e com alguns dos dados que mockamos anteriormente
  expect(res.json).toHaveBeenCalledTimes(1);
  expect(res.json).toHaveBeenCalledWith(campos);
});
```

Como estamos verificando se as funções foram chamadas uma única vez e com os parâmetros corretos, podemos simplificar nossas asserções e utilizar somente uma, em vez de executar `toHaveBeenCalledTimes` e `toHaveBeenCalledWith`. Cada função que "mockamos" com o Jest possui alguns dados disponíveis em `.mock.calls`.

Faça o seguinte `console.log` no seu teste:

```
console.log(res.json.mock.calls);
console.log(res.cookie.mock.calls);
```

Você verá que o log será um array, contendo outros arrays dentro. Esse array principal contém as chamadas da função, já cada um dos arrays menores possui os valores que as funções receberam quando foram executadas. Com isso, poderíamos fazer uma asserção diretamente nesses valores. Caso você prefira, pode trocar os trechos a seguir:

```
  // pode trocar estas asserções
  expect(findUser.usernameAndPassword).toHaveBeenCalledTimes(1);

expect(findUser.usernameAndPassword).toHaveBeenCalledWith(req.body.usernam
e, req.body.password);
  // por esta
  expect(findUser.usernameAndPassword.mock.calls).toEqual([
    [req.body.username, req.body.password]
  ]);

  // pode trocar estas asserções
  expect(res.cookie).toHaveBeenCalledTimes(1);
  expect(res.cookie).toHaveBeenCalledWith('uid', uid);
  // por esta
  expect(res.cookie.mock.calls).toEqual([
    ['uid', uid]
  ]);

  // e também pode trocar estas asserções
  expect(res.json).toHaveBeenCalledTimes(1);
  expect(res.json).toHaveBeenCalledWith({ uid });
```

```
  // por esta
   expect(res.json.mock.calls).toEqual([
    [{ uid }]
  ]);
```

É algo totalmente opcional e é uma forma diferente de validar as chamadas de seus mocks. Particularmente, prefiro utilizar `toHaveBeenCalledTimes` e `toHaveBeenCalledWith`, pois acho que deixam a leitura dos testes menos subjetiva e mais clara, porém, utilizar `.mock.calls` é uma opção interessante de se conhecer também caso você precise utilizar ou encontre em algum teste por aí.

Testando cenário de usuário não encontrado

Vamos para o nosso cenário de falha, onde o usuário não é encontrado e nosso objeto de resposta é chamado com o status 404 e com o erro disparado. Vamos criar esse novo caso de teste, bem como os mocks de `req` e `res`. No entanto, dessa vez nosso objeto `res` deverá conter a função `status` também:

```
// novo caso de teste
it('Dispara um erro e retorna 404 caso o usuário não seja encontrado',
async () => {
  // objeto req
  const req = {
    body: {
      username: 'usuario',
      password: 'senha-super-secreta'
    }
  };

  // objeto res
  const res = {
    json: jest.fn(() => res),
    cookie: jest.fn(() => res),
    status: jest.fn(() => res) // nova função status
  };
});
```

Podemos simular que a Promise utilizada em `findUser.usernameAndPassword` será rejeitada com algum erro qualquer:

```
it('Dispara um erro e retorna 404 caso o usuário não seja encontrado',
async () => {
  const req = {
    body: {
      username: 'usuario',
      password: 'senha-super-secreta'
    }
  };

  const res = {
    json: jest.fn(() => res),
    cookie: jest.fn(() => res),
    status: jest.fn(() => res)
  };

  // simulamos o erro
  const erro = 'usuário não existente';
  findUser.usernameAndPassword.mockRejectedValueOnce(erro);
});
```

Agora, só precisamos chamar nosso controller e passar os objetos `req` e `res` novamente. Feito isso, vamos realizar as asserções corretas. Também precisamos verificar se `logger.error` foi chamada corretamente:

```
it('Dispara um erro e retorna 404 caso o usuário não seja encontrado',
async () => {
  const req = {
    body: {
      username: 'usuario',
      password: 'senha-super-secreta'
    }
  };

  const res = {
    json: jest.fn(() => res),
    cookie: jest.fn(() => res),
    status: jest.fn(() => res)
```

```
  };

  const erro = 'usuário não existente';
  findUser.usernameAndPassword.mockRejectedValueOnce(erro);

  // executamos o controller
  await authenticate(req, res);

  // verificamos se a função findUser.usernameAndPassword foi chamada
  // uma única vez e com os dados de usuario/senha
  expect(findUser.usernameAndPassword).toHaveBeenCalledTimes(1);

expect(findUser.usernameAndPassword).toHaveBeenCalledWith(req.body.usernam
e, req.body.password);

  // verificamos se logger.error foi executada
  // uma vez e com uma mensagem qualquer
  // mas com o erro disparado
  expect(logger.error).toHaveBeenCalledTimes(1);
  expect(logger.error).toHaveBeenCalledWith(expect.any(String), erro);

  // verificamos se res.status foi executada
  // uma vez e com o valor 404
  expect(res.status).toHaveBeenCalledTimes(1);
  expect(res.status).toHaveBeenCalledWith(404);

  // verificamos se res.json foi executada
  // uma vez e com o texto do erro
  expect(res.json).toHaveBeenCalledTimes(1);
  expect(res.json).toHaveBeenCalledWith(erro);
});
```

Opa, tivemos um pequeno erro. Aparentemente, nosso segundo teste está dizendo que `findUser.usernameAndPassword` foi chamada duas vezes em vez de uma. Portanto, precisamos limpar as chamadas dessa função. Podemos fazer isso com um `afterEach`, como já conhecemos:

```
// antes dos testes
afterEach(() => {
```

```
  jest.clearAllMocks();
});
```

8.4 Aplicando factories para a criação dos objetos dos testes

Você deve ter notado que nossos códigos de `req` e `res` são praticamente idênticos, certo? Precisaremos criar esses objetos várias e várias vezes nos nossos testes. Podemos criar uma função e deixá-la responsável apenas pela criação desses valores, o que facilitará nosso trabalho e reduzirá algumas linhas de código.

Vamos criar uma pasta `utils` e, dentro dela, um arquivo chamado `create.js` , na raiz do projeto. Nesse arquivo, vamos criar e exportar três funções, uma chamada `createReq` , uma chamada `createRes` e outra chamada `z` . Por enquanto, elas retornarão apenas objetos vazios:

```
// arquivo utils/create.js

// exportamos createReq que cria e retorna um objeto vazio
export const createReq = () => {
  const req = {};
  return req;
};

// exportamos createRes que cria e retorna um objeto vazio
export const createRes = () => {
  const res = {};
  return res;
};

// exportamos createAuth que cria e retorna um objeto vazio
export const createAuth = () => {
  const auth = {};
```

```
  return auth;
};
```

Criando factory de requisições

Podemos simular nossos objetos de `req` e `res` e também os dados de um usuário, diretamente nessas funções.

A começar pela função `createReq`, até o momento só utilizamos seu campo `body`. Então vamos criar um objeto vazio para esse campo.

```
// criamos um campo body vazio
export const createReq = () => {
  const req = {
    body: {}
  };
  return req;
};

export const createRes = () => {
  const res = {};
  return res;
};

export const createAuth = () => {
  const auth = {};
  return auth;
};
```

Criando factory de respostas

Na função `createRes`, já utilizamos até o momento os métodos `status`, `json` e `cookie`. Vamos criá-los agora:

```
export const createReq = () => {
  const req = {};
  return req;
};

// criamos os mocks de json, cookie e status
```

```
export const createRes = () => {
  const res = {
    json: jest.fn(() => res),
    cookie: jest.fn(() => res),
    status: jest.fn(() => res)
  };
  return res;
};

export const createAuth = () => {
  const auth = {};
  return auth;
};
```

Criando factory de autenticação

Para a função `createAuth`, podemos contar com a ajuda do utilitário `faker`, o mesmo que usamos no projeto anterior para gerar os campos de `username` e `password` aleatoriamente para nós. Vamos importar esse utilitário e criar os valores no objeto que retornamos dessa função:

```
// importamos o pacote faker
import faker from 'faker';

export const createReq = () => {
  const req = {};
  return req;
};

export const createRes = () => {
  const res = {
    json: jest.fn(() => res),
    cookie: jest.fn(() => res),
    status: jest.fn(() => res)
  };
  return res;
};

export const createAuth = () => {
```

```
  // geramos um usuario e senha
  const auth = {
    username: faker.internet.userName(),
    password: faker.internet.password(),
  };
  return auth;
};
```

Adaptando as factories para receber valores extras

Só falta um ajuste para que nossas funções fiquem flexíveis para nossa utilização. É normal que em alguns cenários nós precisemos passar dados extras para customizar (dentro do próprio teste) um objeto de requisição ou resposta.

Para isso, vamos receber então, em cada uma das funções, um parâmetro chamado `extra`, e agrupá-lo com os objetos que criamos:

```
import faker from 'faker';

// recebemos o parâmetro extra
export const createReq = (extra) => {
  const req = {
    body: {},
    ...extra // fazemos spread e agrupamos no objeto criado
  };
  return req;
};

// recebemos o parâmetro extra
export const createRes = (extra) => {
  const res = {
    json: jest.fn(() => res),
    cookie: jest.fn(() => res),
    status: jest.fn(() => res),
    ...extra // fazemos spread e agrupamos no objeto criado
  };
  return res;
};
```

```
// recebemos o parâmetro extra
export const createAuth = (extra) => {
  const auth = {
    username: faker.internet.userName(),
    password: faker.internet.password(),
    ...extra // fazemos spread e agrupamos no objeto criado
  };
  return auth;
};
```

Com isso, poderemos inserir novos campos nesses objetos quando necessário e até mesmo sobrescrever qualquer um existente. Agora também temos funções que vão construir esses objetos para nós sempre que precisarmos.

Esse padrão que acabamos de aplicar (que utiliza uma função que cria um objeto) se chama `factory` (ou *fábrica*, em português). É comum chamar esse cenário de *test object factory*, já que criamos funções para criar objetos para os nossos testes.

Utilizando as factories no projeto

Vamos inserir esse arquivo no nosso `jest.config` para que possamos importar de forma simplificada:

```
export default {
  setupFiles: [
    '<rootDir>/jest.setup.js'
  ],
  moduleNameMapper: {
    '^controllers/(.*)$': '<rootDir>/src/controllers/$1',
    '^middlewares/(.*)$': '<rootDir>/src/middlewares/$1',
    '^services/(.*)$': '<rootDir>/src/services/$1',
    '^utils/(.*)$': '<rootDir>/utils/$1',
  }
}
```

Vamos voltar aos testes que criamos para o controller de autenticação e importar as funções desse módulo:

```
// arquivo __tests__/controllers/auth.controller.unit.js
import { createReq, createRes, createAuth } from 'utils/create';
```

E vamos utilizar essas funções em vez de criar os objetos de `req` e `res` manualmente. No primeiro cenário de teste, vamos substituir o objeto `req` pela função `createReq`. Também passaremos como parâmetro um novo objeto de `body`, contendo o retorno da função `createAuth`:

```
it('Encontra o usuário e insere seu UID em cookie', async () => {
  // executamos createReq e passamos como body
  // o retorno de createAuth
  const req = createReq({ body: createAuth() });

  // restante do código omitido
```

Vamos fazer o mesmo para o objeto `res`:

```
it('Encontra o usuário e insere seu UID em cookie', async () => {
  const req = createReq({ body: createAuth() });
  // executamos createRes
  const res = createRes();

  // restante do código omitido
```

E o mesmo para o teste de erro:

```
it('Dispara um erro e retorna 404 caso o usuário não seja encontrado',
async () => {
  const req = createReq({ body: createAuth() });
  const res = createRes();

  // restante do código omitido
```

Bem mais limpo e organizado, não acha?

A lógica que aplicamos para testar qualquer controller é esta:

- Criamos um objeto de requisição;
- Criamos um objeto de resposta;
- Realizamos os mocks dos módulos necessários;

- Executamos o controller fornecendo os objetos de requisição e resposta.

Não tem mistério. Com isso, deixo para você o desafio de realizar os testes dos controllers `user.controller.js` e `users.controller.js`. É só aplicar essa mesma sequência de passos que você conseguirá sem problemas. Caso precise criar alguma função para auxiliar na criação dos seus objetos de testes, sinta-se à vontade. Lembre-se de que todo o código-fonte testado está no repositório que contém todos os projetos (https://javascriptassertivo.com.br/).

Chegou a hora de testar outra camada importante da nossa aplicação Express: os middlewares.

8.5 Testando middlewares

Um middleware, no Express, é bem parecido com o controller que testamos. É uma função que recebe os objetos com a requisição e com a resposta (`req` e `res`), e também recebe um terceiro, comumente chamado de `next`.

Esse parâmetro serve para que a sequência dos middlewares (sua cadeia) continue executando normalmente: exatamente igual ao diagrama sobre cadeia de responsabilidades que vimos quando testamos a CLI.

Caso algum parâmetro seja enviado ao executar a função `next`, a cadeia é interrompida e os demais middlewares não serão executados, pois o Express entenderá que esse parâmetro identifica um erro, o que fará com que somente os middlewares de tratamento de erro sejam executados.

Podemos pensar que um middleware segue mais ou menos este formato:

```
// recebe os 3 parâmetros
const middleware1 = (req, res, next) => {
  // realiza qualquer operação
  next(); // prossegue execução dos próximos middlewares
};
```

Ao executar a função `next`, o próximo middleware será executado (caso esteja devidamente registrado na aplicação). E, caso algo seja fornecido para a função `next`, os demais serão suspensos:

```
// recebe os 3 parâmetros
const middleware1 = (req, res, next) => {
  // realiza qualquer operação
  next('Ocorreu algum erro'); // prossegue execução dos próximos
middlewares
};
```

Testando nosso primeiro middleware do Express

Com isso em mente, podemos testar nosso primeiro middleware. Testaremos a função `getUserData`, no arquivo `middlewares/user.middleware.js`, responsável por coletar os dados do usuário que está disparando a requisição. Dê uma analisada nele antes de prosseguirmos.

Vamos criar essa mesma pasta dentro do diretório `__tests__` e criar nosso arquivo `user.middleware.unit.js`. Aproveite para já importar a função lá:

```
// arquivo __tests__/middlewares/user.middleware.unit.js
import { getUserData } from 'middlewares/user.middleware';
```

Como esse arquivo possui vários middlewares e, por ora, testaremos somente esse, vamos criar um bloco com um `describe`, indicando a função que testaremos:

```
import { getUserData } from 'middlewares/user.middleware';

// criamos o describe
describe('getUserData encontra as informações a partir de um uid', () => {
```

```
});
```

Para testar esse middleware, precisaremos realizar um mock do módulo `services/user/find`, já que a função `find.uid` é executada por esse middleware. Chamaremos o mock de `service`:

```
import { getUserData } from 'middlewares/user.middleware';

// Realizamos o mock e importamos tudo do service
import service from 'services/user/find';
jest.mock('services/user/find');

describe('getUserData encontra as informações a partir de um uid', () => {

});
```

Vamos criar nosso primeiro caso de teste para esse middleware. Testaremos o cenário onde a função `find.uid` (ou `service.uid`, já que a renomeamos ao importar) é executada e o usuário é encontrado. Após encontrar o usuário, esse middleware insere no objeto `req` os dados dentro da chave `user`. Com isso, podemos ter o seguinte título para nosso teste:

```
// Middleware
import { getUserData } from 'middlewares/user.middleware';

// Mock service
import service from 'services/user/find';
jest.mock('services/user/find');

describe('getUserData encontra as informações a partir de um uid', () => {
  it('Insere as informações encontradas no objeto da requisição', async ()
=> {

  });
});
```

Como precisaremos criar os objetos de `req` e `res`, vamos aproveitar e importar isso dos nossos utilitários:

```
import { getUserData } from 'middlewares/user.middleware';

// importamos os utilitários
import { createReq, createRes, createUser } from 'utils/create';

import service from 'services/user/find';
jest.mock('services/user/find');

describe('getUserData encontra as informações a partir de um uid', () => {
  it('Insere as informações encontradas no objeto da requisição', async ()
=> {

  });
});
```

Como também precisaremos gerar um usuário, podemos importar a seguinte função `createUser` do nosso pacote da CLI:

```
import { getUserData } from 'middlewares/user.middleware';

// importamos a função createUser
import { createUser } from '@jsassertivo/cli/commands/user';
import { createReq, createRes } from 'utils/create';


import service from 'services/user/find';
jest.mock('services/user/find');

describe('getUserData encontra as informações a partir de um uid', () => {
  it('Insere as informações encontradas no objeto da requisição', async ()
=> {

  });
});
```

Caso você prefira, você pode inserir essa função no arquivo `utils/create` também, como eu fiz no repositório que contém o código-fonte desses projetos.

Já podemos iniciar nosso primeiro teste. Vamos criar nossos objetos de `req`, `res` e também os dados de um usuário `user`:

```
describe('getUserData encontra as informações a partir de um uid', () => {
  it('Insere as informações encontradas no objeto da requisição', async ()
=> {
    const user = createUser(); // criamos usuário
    const req = createReq(); // criamos requisição
    const res = createRes(); // criamos resposta
  });
});
```

Para que o middleware encontre o usuário, o `uid` deve ser fornecido no campo `cookies` da requisição. Vamos inserir o `uid` do usuário que criamos:

```
describe('getUserData encontra as informações a partir de um uid', () => {
  it('Insere as informações encontradas no objeto da requisição', async ()
=> {
    const user = createUser();
    // inserimos uid na requisição
    const req = createReq({ cookies: { uid: user.uid } });
    const res = createRes();
  });
});
```

Precisamos fazer o mock do retorno da função `service.uid` para que ela retorne o usuário que criamos:

```
describe('getUserData encontra as informações a partir de um uid', () => {
  it('Insere as informações encontradas no objeto da requisição', async ()
=> {
    const user = createUser();
    const req = createReq({ cookies: { uid: user.uid } });
    const res = createRes();

    // mockamos o valor resolvido da função
    service.uid.mockResolvedValueOnce(user);
  });
});
```

Pronto, podemos executar nosso middleware passando os objetos que criamos. Vamos criar nossas asserções para verificar se a função `service.uid` será executada e também nos assegurar de que,

após executar o middleware, o objeto `req` terá os dados do usuário dentro de `req.user`:

```
describe('getUserData encontra as informações a partir de um uid', () => {
  it('Insere as informações encontradas no objeto da requisição', async ()
=> {
    const user = createUser();
    const req = createReq({ cookies: { uid: user.uid } });
    const res = createRes();

    service.uid.mockResolvedValueOnce(user);

    // executamos o middleware
    await getUserData(req, res);

    // verificamos se a função service.uid
    // foi executada uma vez com o uid do usuário
    expect(service.uid).toHaveBeenCalledTimes(1);
    expect(service.uid).toHaveBeenCalledWith(user.uid);

    // verificamos se req.user
    // agora é igual ao usuário
    expect(req.user).toEqual(user);
  });
});
```

Opa, temos um erro! Recebemos esse erro porque não passamos o parâmetro `next`, que é justamente a função responsável por continuar os middlewares ou interromper a cadeia de execução. Vamos criar uma função mock do Jest e passá-la como parâmetro. Também podemos aproveitar e garantir que essa função será chamada uma única vez e sem receber nada como parâmetro:

```
describe('getUserData encontra as informações a partir de um uid', () => {
  it('Insere as informações encontradas no objeto da requisição', async ()
=> {
    const user = createUser();
    const req = createReq({ cookies: { uid: user.uid } });
    const res = createRes();
    const next = jest.fn(); // criamos uma função mock qualquer
```

```
    service.uid.mockResolvedValueOnce(user);

    // passamos como parâmetro next
    await getUserData(req, res, next);

    expect(service.uid).toHaveBeenCalledTimes(1);
    expect(service.uid).toHaveBeenCalledWith(user.uid);

    expect(req.user).toEqual(user);

    // verificamos se foi chamada uma única vez
    expect(next).toHaveBeenCalledTimes(1);
    expect(next).toHaveBeenCalledWith();
  });
});
```

Podemos mover a criação dessa função para nosso utilitário dentro de `utils/create`, criando uma função `createNext` lá:

```
export const createNext = () => {
  const next = jest.fn();

  return next;
};
```

Assim, podemos importá-la:

```
// adicionamos o import da função createNext
import { createReq, createRes, createNext } from 'utils/create';
```

E dentro do nosso test utilizamos ela:

```
const next = createNext();
```

Com isso, seguimos um mesmo padrão para a criação desses objetos básicos dos testes.

Testando o cenário de erro do middleware de usuários

Podemos criar um caso de teste para o cenário de erro. Vamos criar um novo `it` para isso dentro do `describe` que já temos:

```
describe('getUserData encontra as informações a partir de um uid', () => {
  // primeiro it omitido

  it('Retorna uma mensagem de erro caso não encontre', async () => {

  });
});
```

Esse cenário será bem parecido com o anterior. Então vamos criar a mesma estrutura dos objetos `user`, `req`, `res` e `next`:

```
it('Retorna uma mensagem de erro caso não encontre', async () => {
  const user = createUser();
  const req = createReq({ cookies: { uid: user.uid } });
  const res = createRes();
  const next = createNext();
});
```

Agora precisamos simular o cenário de erro da função `service.uid`. Vamos criar um objeto de erro para utilizar nesse cenário (se quiser, você pode criar uma função `createError` em `utils/create` para isso também):

```
it('Retorna uma mensagem de erro caso não encontre', async () => {
  const user = createUser();
  const req = createReq({ cookies: { uid: user.uid } });
  const res = createRes();
  const next = createNext();
  // criamos objeto de erro
  const error = new Error('Erro ao consultar os dados');

  // simula cenário da Promise rejeitada com o erro criado
  service.uid.mockRejectedValueOnce(error);
});
```

E podemos executar novamente nosso middleware e realizar as asserções necessárias:

```
 it('Retorna uma mensagem de erro caso não encontre', async () => {
    const user = createUser();
    const req = createReq({ cookies: { uid: user.uid } });
    const res = createRes();
    const next = createNext();
    const error = new Error('Erro ao consultar os dados');

    service.uid.mockRejectedValueOnce(error);

    // executamos o middleware
    await getUserData(req, res, next);

    // verificamos se a função service.uid
    // foi chamada uma vez e com o uid do usuáro
    expect(service.uid).toHaveBeenCalledTimes(1);
    expect(service.uid).toHaveBeenCalledWith(user.uid);

    // verificamos se a função res.status
    // foi chamada uma vez e com o status de 401
    expect(res.status).toHaveBeenCalledTimes(1);
    expect(res.status).toHaveBeenCalledWith(401);

    // verificamos se a função res.json
    // foi chamada uma vez e com uma mensagem de erro qualquer
    expect(res.json).toHaveBeenCalledTimes(1);
    expect(res.json).toHaveBeenCalledWith({ message: expect.any(String)
});

    // verificamos se a função next
    // foi chamada uma vez e com o erro criado
    expect(next).toHaveBeenCalledTimes(1);
    expect(next).toHaveBeenCalledWith(error);
  });
```

Provavelmente receberemos o erro pelo fato de a função `service.uid` ser chamada novamente e acumularmos sua execução. O ideal é zerarmos a contagem das execuções dessas funções para que nossos testes fiquem previsíveis e que sejam executados de forma isolada.

Podemos limpar essas execuções inserindo o `afterEach` como já fizemos algumas vezes anteriormente. Dessa vez, vamos colocar dentro do nosso bloco de `describe`:

```
describe('getUserData encontra as informações a partir de um uid', () => {
  // inserimos afterEach limpando os mocks
  afterEach(() => {
    jest.clearAllMocks();
  });

  // testes omitidos
});
```

E pronto, nossos testes passarão e acabamos de testar nosso primeiro middleware!

O processo para testar os demais middlewares será exatamente o mesmo que fizemos agora, então deixo mais esse desafio para você para que possamos dar o próximo passo e testar outra camada da nossa aplicação: a de *services*.

8.6 Testando services

O conteúdo entendido como *services* dentro dessa aplicação é, basicamente, uma camada intermediária entre a própria aplicação Express e as funções da CLI que vimos na parte anterior deste livro. Como boa parte das regras de negócio que estamos testando foram desenvolvidas no projeto anterior, conseguimos ter um nível de reúso de código bem alto!

A grande maioria dos arquivos simplesmente exporta as funções da CLI. Algumas funções são até renomeadas nessas exportações com um nome que faça mais sentido para a aplicação.

Por exemplo:

```
// arquivo services/users/find.js
export { loadDatabase as findAll } from
'@jsassertivo/cli/src/database/file.js';

// arquivo services/user/create.js
export { createUser } from '@jsassertivo/cli/src/database/user/create.js'

// arquivo services/user/create.js
export { createUser } from '@jsassertivo/cli/src/database/user/create.js'

// arquivo services/user/remove.js
export { removeUser } from '@jsassertivo/cli/src/database/user/remove.js';

// arquivo services/user/update.js
export { updateUserByUid } from
'@jsassertivo/cli/src/database/user/update.js'
```

No entanto, essa API que estamos testando possui algumas regras de negócio específicas presentes no arquivo `services/user/find.js`, já que é possível consultar usuários por diferentes campos, como `uid`, `username` e `email`. Essas regras de consulta são aplicadas diretamente nos métodos desse `service` em específico que vamos testar agora.

Testando os serviços de usuário: consulta por username

Com isso, vamos criar dentro do diretório `__tests__` a pasta `services/user` e, dentro dela, nosso arquivo `find.unit.js`. Vamos iniciar pela função `getUserByUsername`, então vamos aproveitar para importá-la e também criar nosso bloco com `describe` / `it`:

```
// arquivo __tests__/services/user/find.unit.js

// importamos a função que testaremos
import { getUserByUsername } from 'services/user/find';

// criamos um describe/it
describe('getUserByUsername encontra um usuário por username', () => {
  it('Retorna um usuário ao encontrar', async () => {
```

```
  });
});
```

Para testar essa função precisaremos realizar um mock da função `loadDatabase` exportada pelo arquivo `@jsassertivo/cli/src/database/file.js`, então vamos realizá-lo. Como precisaremos limpar esse mock também, já vamos colocar essa limpeza em um `afterEach`:

```
import { getUserByUsername } from 'services/user/find';

// Importamos e realizamos o mock desse módulo
import { loadDatabase } from '@jsassertivo/cli/src/database/file.js';
jest.mock('@jsassertivo/cli/src/database/file.js');

afterEach(() => {
  jest.clearAllMocks();
});

describe('getUserByUsername encontra um usuário por username', () => {
  it('Retorna um usuário ao encontrar', async () => {

  });
});
```

Para que essa função retorne uma lista de usuários, precisaremos criar essa lista. Sugiro que você crie algum método, como `createUserList` dentro de `utils/create`. Caso prefira, você pode importar o método que usamos anteriormente para realizar essa tarefa:

```
import { getUserByUsername } from 'services/user/find';

// importamos a função que realizará a criação do usuário
import { createUser } from '@jsassertivo/cli/commands/user';

import { loadDatabase } from '@jsassertivo/cli/src/database/file.js';
jest.mock('@jsassertivo/cli/src/database/file.js');

afterEach(() => {
```

```
  jest.clearAllMocks();
});

describe('getUserByUsername encontra um usuário por username', () => {
  it('Retorna um usuário ao encontrar', async () => {

  });
});
```

Vamos criar uma lista de usuários para que possamos trabalhar no teste. Será uma lista com três usuários e usaremos desestruturação para consultar o primeiro a fim de tentar encontrar esse registro no nosso teste:

```
describe('getUserByUsername encontra um usuário por username', () => {
  it('Retorna um usuário ao encontrar', async () => {
    // lista de usuários com alguns registros
    const userList = [
      createUser(),
      createUser(),
      createUser(),
    ];
    // primeiro usuário da lista
    const [user] = userList;
  });
});
```

E vamos indicar que essa lista de usuários será o retorno do método `loadDatabase`, que mockamos anteriormente:

```
describe('getUserByUsername encontra um usuário por username', () => {
  it('Retorna um usuário ao encontrar', async () => {
    const userList = [
      createUser(),
      createUser(),
      createUser(),
    ];
    const [user] = userList;

    // indicamos que o retorno será userList
    loadDatabase.mockResolvedValueOnce(userList);
```

```
  });
});
```

Tudo o que precisamos fazer é executar a função `getUserByUsername` fornecendo o username do primeiro usuário que desestruturamos e criar nossas asserções com base no retorno:

```
describe('getUserByUsername encontra um usuário por username', () => {
  it('Retorna um usuário ao encontrar', async () => {
    const userList = [
      createUser(),
      createUser(),
      createUser(),
    ];
    const [user] = userList;

    loadDatabase.mockResolvedValueOnce(userList);

    // executamos a função informando user.userName
    // e armazenamos o usuário de retorno
    const retorno = await getUserByUsername(user.userName);

    // verificamos se o retorno é o usuário
    expect(retorno).toEqual(user);
  });
});
```

Testando erros na consulta por username

Com isso, temos o nosso teste para o cenário de sucesso funcionando. Vamos criar um `it` dentro desse mesmo `describe` para o cenário de erro:

```
it('Retorna um erro caso o usuário não seja encontrado', async () => {

});
```

Precisaremos seguir o mesmo processo: criar uma lista de usuários e executar a nossa função. No entanto, dessa vez, o usuário que tentaremos encontrar tem que ser um que não está na base. Dessa forma, vamos criar um novo usuário em vez de utilizar um da lista através de desestruturação. Como também testaremos um cenário de erro, vamos colocar nosso `try/catch`:

```
it('Retorna um erro caso o usuário não seja encontrado', async () => {
   try { // colocamos nosso bloco try
     const userList = [ // criamos uma nova lista de usuariios
       createUser(),
       createUser(),
       createUser(),
     ];
     // criamos um novo, sem utilizar da lista
     const user = createUser();
   } catch (err) { // bloco catch

   }
 });
```

Como vimos no começo da nossa jornada, também é interessante indicar quantas asserções teremos para esse teste de falha usando `expect.assertions`. No caso, teremos somente uma:

```
it('Retorna um erro caso o usuário não seja encontrado', async () => {
   // indicamos quantas asserções teremos
   expect.assertions(1);

   try {
     const userList = [
       createUser(),
       createUser(),
       createUser(),
     ];
     const user = createUser();
   } catch (err) {

   }
 });
```

Vamos indicar que a variável `userList` será o valor resolvido da Promise `loadDatabase`:

```
it('Retorna um erro caso o usuário não seja encontrado', async () => {
    expect.assertions(1);

    try {
      const userList = [
        createUser(),
        createUser(),
        createUser(),
      ];
      const user = createUser();

      loadDatabase.mockResolvedValueOnce(userList);
    } catch (err) {

    }
  });
```

Com isso, podemos executar a função `getUserByUsername` informando o username do usuário que criamos e já colocar nossa asserção dentro do `catch`. Nesse cenário, podemos validar somente que o `err` será um objeto do tipo `Error`:

```
 it('Retorna um erro caso o usuário não seja encontrado', async () => {
    expect.assertions(1);

    try {
      const userList = [
        createUser(),
        createUser(),
        createUser(),
      ];
      const user = createUser();

      loadDatabase.mockResolvedValueOnce(userList);

      // executamos a função informando o nome de usuário
      await getUserByUsername(user.userName);
    } catch (err) {
```

```
      // asseguramos que temos um erro recebido
      expect(err).toEqual(expect.any(Error));
    }
  });
```

E pronto! Temos nosso `service` devidamente testado.

Testando utilitários dos serviços

Vamos testar o utilitário `basedOnQuery`. O que ele faz é, baseado no objeto recebido (`query`), retornar um nome de uma função (do próprio `service`) a ser executada, assim como o parâmetro dessa própria query necessária. Caso não encontre nenhuma função, um erro é retornado. Ou seja, temos algo mais ou menos assim:

- Query com uid: retorna o nome `uid` e também o valor do `uid`.
- Query com username: retorna o nome `username` e também o valor do `username`.
- Query com e-mail: retorna o nome `email` e também o valor do `email`;
- Caso não encontre nenhuma dessas três, retorna um erro.

Criaremos nosso `describe` para essa função. Vamos começar com o cenário de `uid` e já vamos criar um `it` para isso:

```
// criamos o describe
describe('basedOnQuery encontra uma função dentro de service', () => {
  // criamos o it
  it('Retorna a função que consulta usuário por uid e seu valor', () => {
  });
});
```

Precisamos criar uma variável com o nome da função que teremos de retorno e a query que será fornecida para a função `basedOnQuery`:

```
describe('basedOnQuery encontra uma função dentro de service', () => {
  it('Retorna a função que consulta usuário por uid e seu valor', () => {
    // criamos a função que esperamos
    const funcao = 'uid';
    // criamos a query
```

```
    const query = { uid: 'qualquer-uid' };
    // executamos a função com a query que criamos
    const { by, param } = basedOnQuery(query);
  });
});
```

Vamos criar nossas asserções para verificar que os valores de `by` e `param`, retornados pela função, são iguais aos que esperamos em `funcao` e `query[by]` (já que `by` é a chave do objeto `query` que montamos):

```
describe('basedOnQuery encontra uma função dentro de service', () => {
  it('Retorna a função que consulta usuário por uid e seu valor', () => {
    const funcao = 'uid';
    const query = { uid: 'qualquer-uid' };
    const { by, param } = basedOnQuery(query);

    // criamos nossa asserção para verificar
    // que o valor de by é igual à função
    expect(by).toEqual(funcao);

    // verificamos se o param retornado
    // é igual ao valor que temos na query
    expect(param).toEqual(query[by]);
  });
});
```

Podemos triplicar esse mesmo `it` e adaptar as variáveis para os cenários de `email` e `username`:

```
describe('basedOnQuery encontra uma função dentro de service', () => {
  it('Retorna a função que consulta usuário por uid e seu valor', () => {
    const funcao = 'uid';
    const query = { uid: 'qualquer-uid' };
    const { by, param } = basedOnQuery(query);

    expect(by).toEqual(funcao);
    expect(param).toEqual(query[by]);
  });

  // novo it para pesquisar por email
```

```
  it('Retorna a função que consulta usuário por email e seu valor', () =>
{
    // campo de email
    const funcao = 'email';
    // query com email
    const query = { email: 'qualquer@email' };
    const { by, param } = basedOnQuery(query);

    expect(by).toEqual(funcao);
    expect(param).toEqual(query[by]);
  });

  // novo it para pesquisar por username
  it('Retorna a função que consulta usuário por username e seu valor', ()
=> {
    // campo de username
    const funcao = 'username';
    // query com username
    const query = { username: 'usuario' };
    const { by, param } = basedOnQuery(query);

    expect(by).toEqual(funcao);
    expect(param).toEqual(query[by]);
  });
});
```

Criando uma "tabela" para simplificar nossos testes semelhantes

Se você notou, nossos três testes são praticamente iguais, só mudando os parâmetros. Existe um utilitário que pode ajudar bastante a manter a limpeza dos nossos testes para esse caso: a função `it.each`. Essa função recebe um array de arrays como parâmetro e retorna uma outra, que deve ser executada para cada teste, mais ou menos assim:

```
it.each([
  [], // cenário de teste 1
  [], // cenário de teste 2
  [], // cenário de teste 3
```

```
])('Titulo do teste', () => {
  // teste
});
```

Conseguimos fornecer dentro dos arrays os valores que queremos executar para cada um de nossos testes:

```
 it.each([
    ['uid', { uid: 'qualquer-uid' }], // teste de uid
    ['email', { email: 'qualquer@email.com' }], // teste de email
    ['username', { username: 'usuario' }], // teste de username
  ])('Titulo do teste', () => {
  });
```

Esses valores serão passados para a função de callback que fornecemos após o título do teste. Então podemos receber a função e a query diretamente lá:

```
it.each([
  ['uid', { uid: 'qualquer-uid' }],
  ['email', { email: 'qualquer@email.com' }],
  ['username', { username: 'usuario' }],
])('Titulo do teste', (fn, query) => { // recebemos fn e query nessa
função
});
```

Agora podemos executar nosso teste e fornecer esses valores, diretamente:

```
it.each([
  ['uid', { uid: 'qualquer-uid' }],
  ['email', { email: 'qualquer@email.com' }],
  ['username', { username: 'usuario' }],
])('Titulo do teste', (fn, query) => {
  const { by, param } = basedOnQuery(query);

  expect(by).toEqual(fn);
  expect(param).toEqual(query[by]);
});
```

E, para finalizar, podemos adaptar os títulos dos nossos testes para ficar mais claro o que queremos testar. Podemos utilizar o placeholder `%s` no título para imprimir uma string, que, no caso, será o próprio nome da `fn` que declaramos:

```
it.each([
  ['uid', { uid: 'qualquer-uid' }],
  ['email', { email: 'qualquer@email.com' }],
  ['username', { username: 'usuario' }],
  // atualizamos o título abaixo
])('Retorna a função que consulta usuário por %s e seu valor', (fn, query)
=> {
  const { by, param } = basedOnQuery(query);

  expect(by).toEqual(fn);
  expect(param).toEqual(query[by]);
});
```

Dessa forma, teremos os três títulos:

- Retorna a função que consulta usuário por uid e seu valor.
- Retorna a função que consulta usuário por e-mail e seu valor.
- Retorna a função que consulta usuário por username e seu valor.

Caso precisássemos informar mais algum valor no título, precisaríamos somente inseri-lo nos arrays dos testes e adaptar o título com os novos valores. A leitura fica muito mais fácil e com bem menos repetição de código, não acha?

Como um último teste, vamos testar o cenário de falha. Vamos começar criando mais um `it` :

```
it('Dispara uma mensagem de erro caso não encontre nenhuma das três
funções', () => {
});
```

Para esse cenário, vamos criar um `try/catch` e uma query com qualquer valor para executar nosso teste. Como não estamos

trabalhando com código assíncrono, não precisamos da declaração `expect.assertions` dessa vez:

```
it('Dispara uma mensagem de erro caso não encontre nenhuma das três
funções', () => {
  try { // criamos bloco try
    // criamos uma query qualquer
    const query = { qualquer: 'query' };
  } catch (err) { // criamos bloco catch

  }
});
```

E agora, vamos executar a função `basedOnQuery` com a query que criamos e verificar que teremos um erro no bloco `catch`:

```
it('Dispara uma mensagem de erro caso não encontre nenhuma das três
funções', () => {
  try {
    const query = { qualquer: 'query' };
    // executamos a função basedOnQuery
    // com a query criada
    basedOnQuery(query);
  } catch (err) {
    // verificamos que temos um erro
    expect(err).toEqual(expect.any(Error));
  }
});
```

E o teste desse utilitário está pronto!

A função restante `getUserByEmail` é exatamente igual à função `getUserByUsername`, mas utiliza o `email` em vez do nome do usuário. Deixo o teste dessa função como um desafio final nesta etapa para você.

Com isso, finalizamos nosso degrau de testes unitários em uma aplicação com Node/Express!

8.7 Próxima parada: integração

Com os nossos testes unitários finalizados, vamos utilizar testes de integração para contemplar todas as camadas dessa aplicação juntas, desde o início do servidor, criando nossas rotas, passando pelas validações dos middlewares até chegar em nossos controllers.

Nesse teste, precisaremos executar e disparar requisições diretamente ao servidor da nossa aplicação. Vamos nessa!

Capítulo 9
Testes de integração na API de usuários

Agora que finalizamos os testes unitários de cada camada da nossa API, chegou a hora de verificar se tudo está funcionando corretamente de forma integrada.

Isso quer dizer que, em vez de testar cada pedaço isolado como fizemos anteriormente com os middlewares, services e controllers, testaremos tudo de forma única.

Vale lembrar que, neste capítulo, ainda trabalharemos no projeto do capítulo anterior e aplicaremos somente essa nova camada de testes.

9.1 Configurações iniciais e factory para cliente HTTP

Precisaremos ajustar algumas configurações e criar alguns utilitários nas "factories" que criamos anteriormente dentro de `utils/create`. Teremos que simular uma lista de usuários em praticamente todos os testes e também criar clientes HTTP, já que precisaremos realizar requisições para o servidor da aplicação de dentro de nossos próprios testes.

Antes de tudo, precisamos fazer um pequeno ajuste para que o pacote do cliente HTTP (axios) funcione corretamente. Vamos ajustar o arquivo `jest.config.js` e inserir uma linha com `testEnvironment: 'node'`:

```
export default {
  testEnvironment: 'node', // inserimos essa linha
  setupFiles: [
```

```
    '<rootDir>/jest.setup.js'
  ],
  moduleNameMapper: {
    '^controllers/(.*)$': '<rootDir>/src/controllers/$1',
    '^middlewares/(.*)$': '<rootDir>/src/middlewares/$1',
    '^services/(.*)$': '<rootDir>/src/services/$1',
    '^utils/(.*)$': '<rootDir>/utils/$1',
  }
}
```

Isso dá mais informações ao Jest sobre o ambiente em que os testes estão rodando e evita que tenhamos alguns problemas ao realizar as requisições de que precisaremos. Precisamos declarativamente indicar que nosso teste é executado em um ambiente com `node` porque o padrão para essa configuração é o valor `jsdom`, um pacote que simula o DOM e o navegador do lado do Node. Vamos ver esse outro pacote mais detalhadamente na hora de testar aplicações front-end.

Agora vamos ajustar nossos utilitários dentro de `utils/create`. Vamos importar o `axios`:

```
import axios from 'axios';
```

Feito isso, vamos criar um objeto para nos auxiliar no trabalho com as requisições. Como nossas requisições se dividirão em basicamente dois tipos (autenticadas e não autenticadas), vamos criar duas chaves para esse objeto, uma `create` e uma `authenticate`. Essas chaves receberão funções que já vamos desenvolver:

```
// criamos o objeto
export const clientHTTP = {
    create: () => {}, // chave create
    authenticate: () => {} // chave authenticate
};
```

Essas funções servirão para criar um cliente HTTP (`create`) e para autenticar um cliente já existente (`authenticate`). Como precisaremos iniciar nosso servidor Express para os testes, faz bastante sentido receber um servidor como parâmetro na função `create`. Já a função

`authenticate`, como servirá para autenticar um cliente, faz sentido receber um cliente e um usuário (que será utilizado para a autenticação). Vamos fazer esses ajustes:

```
export const clientHTTP = {
    create: (server) => {}, // recebemos o parâmetro server
    authenticate: (client, user) => {} // recebemos os parâmetros server e
user
};
```

Para a função `create`, precisamos criar um cliente usando o axios. Podemos fazer isso através da função `axios.create`, que recebe como parâmetro um objeto com uma chave `baseURL`, que podemos utilizar para criar uma URL padrão para nossas requisições. É comum também chamar as informações finais da rota de uma API (como `/user`, `/users`, `/login`) de `endpoint`.

Como nosso servidor roda em `localhost` na porta `8080` podemos utilizar uma variável `baseURL` como base para os endpoints que nosso cliente vai acessar, tendo algo como: `http://localhost:8080/api`.

```
export const clientHTTP = {
    create: (server) => {
        // criamos baseURL do servidor
        const baseURL = `http://localhost:8080/api`;
        // criamos o cliente de axios
        const client = axios.create({ baseURL });
    },
    authenticate: (client, user) => {}
};
```

No entanto, como podemos mudar a porta do servidor caso seja necessário, podemos acessar essa porta de dentro da nossa função, em vez de deixar o valor `8080` fixo. Como receberemos nosso servidor como parâmetro, podemos fazer isso através do método `server.address().port`, que nos retornará a porta em que o servidor está sendo executado. Vamos fazer esse ajuste:

```
export const clientHTTP = {
  create: (server) => {
    const { port } = server.address();
    const baseURL = `http://localhost:${port}/api`;
    const client = axios.create({ baseURL });
  },
  authenticate: (client, user) => {},
};
```

Com isso, caso a porta do servidor seja alterada ao criar nossos testes, não teremos nenhum problema.

Para finalizar a criação do nosso cliente, precisamos fazer mais um ajuste: por padrão, qualquer requisição que retornar com um código de erro (como 400, 404, 401) vai disparar um erro em nosso teste.

Como vamos testar os cenários de erro, podemos criar uma função que, mesmo lidando com códigos de erro, prossiga com os nossos testes como o esperado. É uma função bem simples e ela basicamente retorna o valor recebido. Vamos chamá-la de `responseHandler`, já que ela vai lidar com as respostas das nossas chamadas:

```
// criamos a função responseHandler
const responseHandler = response => response;
export const clientHTTP = {
  create: (server) => {
    const { port } = server.address();
    const baseURL = `http://localhost:${port}/api`;
    const client = axios.create({ baseURL });
  },
  authenticate: (client, user) => {},
};
```

Agora, precisamos utilizá-la para os cenários de sucesso e falha de nossas Promises de resposta. Podemos fazer isso com `interceptors` do axios, que são funções que podemos registrar para serem executadas em nossas requisições ou em nossas respostas. Aqui vamos utilizar a função `client.interceptors.response.use`, que recebe

os interceptadores para cenário de sucesso e erro. Vamos adicioná-la:

```
const responseHandler = response => response;
export const clientHTTP = {
  create: (server) => {
    const { port } = server.address();
    const baseURL = `http://localhost:${port}/api`;
    const client = axios.create({ baseURL });
    // adicionamos interceptador para a resposta
    // e utilizamos o mesmo responseHandler para sucesso e erro
    client.interceptors.response.use(responseHandler, responseHandler);
  },
  authenticate: (client, user) => {},
};
```

Na sequência, só precisamos retornar esse cliente criado no fim de nossa função:

```
const responseHandler = response => response;
export const clientHTTP = {
  create: (server) => {
    const { port } = server.address();
    const baseURL = `http://localhost:${port}/api`;
    const client = axios.create({ baseURL });
    client.interceptors.response.use(responseHandler, responseHandler);

    // retornamos o cliente
    return client;
  },
  authenticate: (client, user) => {},
};
```

Nossa função de autenticação será ainda mais fácil. Basicamente precisamos alterar o cliente recebido para possuir, em seus cabeçalhos, um `cookie` com o `uid` do usuário recebido.

Podemos fazer isso através de `client.defaults.headers.cookies` e adicionar o valor necessário. Como vamos manipular esse cliente,

vamos criar uma variável `authenticated` para deixar as coisas mais claras:

```
export const clientHTTP = {
  create: (server) => {
    const { port } = server.address();
    const baseURL = `http://localhost:${port}/api`;
    const client = axios.create({ baseURL });
    client.interceptors.response.use(responseHandler, responseHandler);

    return client;
  },
  authenticate: (client, user) => {
    // cria variável authenticated
    const authenticated = client;
    // insere cookie de uid com o uid do user recebido
    authenticated.defaults.headers.cookie = `uid=${user.uid};`;
  },
};
```

Agora, para que tudo funcione, precisaremos apenas retornar esse cliente autenticado:

```
const responseHandler = response => response;
export const clientHTTP = {
  create: (server) => {
    const { port } = server.address();
    const baseURL = `http://localhost:${port}/api`;
    const client = axios.create({ baseURL });
    client.interceptors.response.use(responseHandler, responseHandler);

    return client;
  },
  authenticate: (client, user) => {
    const authenticated = client;
    authenticated.defaults.headers.cookie = `uid=${user.uid};`;

    // retornamos o client
    return authenticated;
  },
};
```

E pronto, já temos a estrutura para criar e autenticar nossos clientes HTTP. Vamos fazer apenas mais uma função em nosso arquivo `utils/create`.

Como precisaremos criar alguns usuários e também precisaremos testar algumas coisas como administrador, vamos criar uma função `createUserListWithAdmin`, que será responsável por criar uma lista de usuários e atribuir a `ROLE` de administrador a algum deles:

```
// importamos as ROLES
import ROLES from '@jsassertivo/cli/src/constants/roles';

// criamos a função createUserListWithAdmin
export const createUserListWithAdmin = () => {
  // criamos uma lista de usuários
  const list = createUserList();
  // modificamos a role do primeiro para ser ADMIN
  list[0].role = ROLES.ADMIN;
  // retornamos a lista de usuários
  return list;
};
```

Caso você não tenha feito a função `createUserList` ainda, ela basicamente cria um array com dez usuários utilizando o comando da CLI, como fizemos nos testes anteriores:

```
// importamos a função createUser da CLI
// e renomeamos para createUserCLI
import { createUser as createUserCLI } from
'@jsassertivo/cli/commands/user';

// exporta a função createUserCLI com nome de createUser
export const createUser = createUserCLI;

// cria uma lista de usuários (por padrão, com 10)
export const createUserList = (length = 10) => {
  // preenche cada um deles com um usuário
  // utilizando a função createUser para gerar cada um
  const list = Array.from({ length }, createUser);
  // retorna os usuários criados
```

```
  return list;
};
```

Dessa forma, não precisamos ficar criando usuários e manipulando os pacotes da CLI diretamente, podemos utilizar a função `createUser` para criar um único usuário, a função `createUserList` para criar uma lista qualquer de usuários e a função `createUserListWithAdmin` para criar uma lista de usuários na qual o primeiro é do tipo admin. Assim mantemos nosso padrão de factories somente no arquivo `utils/create`.

9.2 Testando as rotas da API

Com essas configurações, já podemos começar os nossos testes. Vamos começar com as rotas do endpoint de `/api/user`.

Testando a busca no endpoint de usuário

Para começar, vamos criar o arquivo do nosso teste. A ideia é que os arquivos com os testes de integração sigam um padrão como `[rota].integration.js` e fiquem dentro da pasta `__tests__` também. Como vamos testar a rota `user`, vamos criar um arquivo `user.route.integration.js`.

Como vamos precisar iniciar nossa aplicação para cada teste integrado, já vamos aproveitar e importar a função que o arquivo `src/index` exporta, responsável por iniciar a aplicação:

```
// arquivo __tests__/user.route.integration.js
// Importamos a função que inicia a aplicação
import app from '../src/index';
```

Vamos importar as factories que vamos utilizar. Precisaremos das funções `createUserListWithAdmin` e `clientHTTP`:

```
import app from '../src/index';

// importamos as factories
import { createUserListWithAdmin, clientHTTP } from 'utils/create';
```

Como vamos precisar simular a nossa base de dados e criar alguns usuários, vamos aproveitar e realizar esses mocks necessários:

```
import app from '../src/index';

import { createUserListWithAdmin, clientHTTP } from 'utils/create';

// importamos database
import { loadDatabase } from '@jsassertivo/cli/src/database/file.js';

// mock de database e do logger (para evitar ficar mostrando mensagens no
console)
jest.mock('@jsassertivo/cli/src/database/file.js');
jest.mock('@jsassertivo/cli/src/utils/logger.js');

// criamos uma lista de usuários com admin
// extraímos os dados do usuário admin
// e indicamos que a função loadDatabase deverá ser
// resolvida com os usuários que criamos
const users = createUserListWithAdmin();
const [admin] = users;
loadDatabase.mockResolvedValue(users);
```

Com essas configurações no nosso arquivo, podemos começar a escrever o nosso teste. Precisaremos iniciar o servidor da aplicação antes de todos os testes, assim como precisaremos encerrá-lo após os nossos testes rodarem. Podemos usar os hooks `beforeAll` e `afterAll` para nos ajudarem com isso.

Para que possamos armazenar os dados do servidor criado, vamos criar uma variável fora desses hooks e modificar seu valor quando necessário:

```
// criamos variável server
let server;
```

```
// criamos a aplicação
// e armazenamos o retorno na variável server
// antes de iniciar os testes
beforeAll(() => {
  server = app();
});

// após finalizar os testes
// encerramos o servidor criado
afterAll(() => {
  server.close();
});
```

Podemos criar mais uma variável e um hook para nos auxiliar. Como precisaremos ficar manipulando um cliente HTTP, vamos criar uma variável `client` para que, antes de cada teste (ou seja, `beforeEach`), seja criado um novo cliente do axios para nos auxiliar. Como esse hook é executado depois do `beforeAll`, mas antes do `afterAll`, embora a ordem no nosso código não faça diferença, vou colocá-lo entre os dois para manter a semântica desses acontecimentos:

```
let server;
beforeAll(() => {
  server = app();
});

// variável client criada
let client;
beforeEach(() => {
  // cria um novo cliente antes de cada teste
  client = clientHTTP.create(server);
});

afterAll(() => {
  server.close();
});
```

Com isso, já temos a configuração necessária para criar e encerrar nosso servidor Express ao executar os testes a partir da variável

`server`. Também já temos uma variável `client`, que armazenará um cliente HTTP para que possamos realizar as requisições.

Agora vamos trabalhar em nossos testes. Como estamos testando a rota `/user` e vamos testar o método `GET`, vamos criar dois `describes`, um indicando a rota e o outro indicando o tipo da requisição:

```
describe('/user', () => {
  describe('GET', () => {});
});
```

Iniciaremos testando uma requisição não autenticada. Vamos tentar consultar usuários e receber um erro. Vamos criar um `it` com essa descrição, não se esqueça de colocar `async` no callback do teste:

```
describe('/user', () => {
  describe('GET', () => {
    it('Tenta consultar usuários sem estar logado e retorna 401', async ()
=> {
    });
  });
});
```

Tudo o que precisamos fazer é disparar uma requisição para a nossa rota `/user` utilizando o método `get` do nosso `client`:

```
it('Tenta consultar usuários sem estar logado e retorna 401', async () =>
{
  const request = await client.get('/user');
});
```

Por padrão, podemos acessar a resposta de uma requisição com erro (como é o nosso caso) a partir da chave `response` do objeto retornado pela Promise do axios, justamente por causa daquele `handler` que criamos, lembra?

Caso a requisição tivesse sido bem-sucedida, os dados poderiam ser acessados diretamente através da chave `data`, direto no objeto

retornado. Portanto, vamos fazer uma desestruturação e verificar se a resposta do erro recebido foi, de fato, com o status 401:

```
it('Tenta consultar usuários sem estar logado e retorna 401', async () =>
{
  const { response } = await client.get('/user');
  expect(response.status).toEqual(401);
});
```

Para executar os testes de integração, você pode executar o comando `npm run test:integration` ou, caso queira deixar os testes rodando com o Jest e de olho nas alterações dos arquivos, pode executar `npm run test:integration:watch` .

Rode qualquer um desses comandos no seu terminal e você verá que nosso primeiro teste integrado está passando!

Agora vamos para o cenário de sucesso. É possível consultar usuários de três maneiras diferentes: por `uid` , por `username` e por `email` . Como teremos esses três casos diferentes, já vamos criar uma nova estrutura utilizando o `it.each` , como aprendemos anteriormente, dentro desse mesmo `describe` :

```
// criamos it.each
it.each([
    ['username'], // teste por username
    ['email'], // teste por email
    ['uid'], // teste por uid
])('Consulta dados por %s ao estar autenticado', async (query) => {
});
```

Dentro de cada um dos pequenos arrays de `it.each` precisaremos fornecer mais algumas coisas ao nosso teste: o campo usuário, que queremos usar no teste da consulta, e o nome do campo nos dados do usuário. Como já temos uma lista de usuários dentro da variável `users` , podemos pegar qualquer valor de lá, por exemplo, o usuário no índice `1` . Não vamos nos esquecer de receber esses valores na função de callback do nosso teste:

```
it.each([
    // passamos o usuário de índice 1 e o campo que desejamos consultar
    ['username', users[1], 'userName'],
    ['email', users[1], 'email'],
    ['uid', users[1], 'uid'],
    // recebemos novos valores na função de callback
])('Consulta dados por %s ao estar autenticado', async (query, user,
field) => {
});
```

A necessidade de passar o campo como um argumento extra vem do fato de, na base de dados e consequentemente nos dados dos usuários, o campo `username` possuir a letra `n` maiúscula. Assim também deixamos o primeiro valor apenas para a query e para o título do teste, o segundo com os dados do usuário e o terceiro apenas com o campo que desejamos consultar.

Agora precisamos autenticar o cliente HTTP para essas requisições. Vamos usar o método `clientHTTP.authenticate` para nos ajudar com essa tarefa. Precisamos fornecer para essa função um cliente do axios já existente e um usuário para que possamos nos autenticar. Já temos em nosso escopo uma variável chamada `admin`, que contém os dados de um usuário administrador (caso não se lembre, definimos essa variável após executar a função `createUserListWithAdmin` ao realizar o retorno do mock da função `loadDatabase`), no início do nosso arquivo de teste.

Portanto, podemos usar as variáveis `client` e `admin` já criadas:

```
it.each([
    ['username', users[1], 'userName'],
    ['email', users[1], 'email'],
    ['uid', users[1], 'uid'],
])('Consulta dados por %s ao estar autenticado', async (query, user,
field) => {
    // executamos clientHTTP.authenticate
    // fornecendo client e admin já criados
    const authenticated = clientHTTP.authenticate(client, admin);
});
```

E podemos usar a variável `authenticated` para disparar uma requisição autenticada. Como temos a consulta, o usuário que desejamos consultar nas variáveis `query`, `user` e `field`, podemos usar esses valores para montar nossa requisição:

```
it.each([
    ['username', users[1], 'userName'],
    ['email', users[1], 'email'],
    ['uid', users[1], 'uid'],
])('Consulta dados por %s ao estar autenticado', async (query, user,
field) => {
    const authenticated = clientHTTP.authenticate(client, admin);
    // criamos requisição usando authenticated.get
    // no endpoint /user com os dados de query/usuario/campo desejados
    const request = await authenticated.get(`/user?
${query}=${user[field]}`);
});
```

Com isso, só precisamos realizar nossas asserções, validando que o código (`status`) da requisição é 200 e que os dados (`data`) retornados são do usuário que consultamos:

```
it.each([
    ['username', users[1], 'userName'],
    ['email', users[1], 'email'],
    ['uid', users[1], 'uid'],
])('Consulta dados por %s ao estar autenticado', async (query, user,
field) => {
    const authenticated = clientHTTP.authenticate(client, admin);
    const request = await authenticated.get(`/user?
${query}=${user[field]}`);

    // criamos asserção que assegura
    // o código 200 na requisição
    expect(request.status).toEqual(200);
    // criamos asserção que assegura
    // que o dado de retorno é igual ao usuário
    expect(request.data).toEqual(user);
});
```

E com isso, testamos nossa requisição do tipo `GET` para esse endpoint de usuário!

Testando a alteração de registros de usuário

Agora vamos testar o método que cria um registro de usuário. Ainda nesse arquivo, vamos criar um novo `describe` para o nosso método `POST`:

```
describe('/user', () => {
  // primeiro teste omitido

  describe('POST', () => {

  });
});
```

Para alterar ou criar um usuário, temos algumas regras extras de validação e alguns cenários mais interessantes, pois a requisição deve:

- Ser feita por uma pessoa autenticada.
- Ser feita por uma pessoa com a role do tipo `ADMIN`.
- Possuir o `uid` do usuário que deseja alterar e os campos novos a serem alterados.

Vamos começar com o cenário onde a requisição não foi autenticada. Devemos receber o mesmo erro 401 do teste anterior. Vamos copiar e colar esse teste para dentro do nosso `describe` fazendo as devidas alterações:

```
describe('POST', () => {
  // copiamos o teste e alteramos o título
  it('Tenta cadastrar usuário sem estar logado e retorna 401', async () =>
{
    // alteramos também o client para usar post ao invés de get
    const { response } = await client.post('/user');
    expect(response.status).toEqual(401);
  });
});
```

A seguir, vamos testar o cenário onde o usuário está autenticado, mas não possui a role de `ADMIN`. Vamos criar nosso novo `it`:

```
it('Tenta cadastrar usuário estando logado mas sem possuir a role ADMIN',
() => {
});
```

Vamos autenticar nosso cliente HTTP, mas usando qualquer usuário da nossa variável `users` sem ser o do índice 0, que é o admin. Após isso, podemos disparar nossa requisição:

```
it('Tenta cadastrar usuário estando logado mas sem possuir a role ADMIN',
() => {
  // autenticamos usuário no índice 1
  const unauthorized = clientHTTP.authenticate(client, users[1]);
  // realizamos a requisição novamente
  const { response } = await unauthorized.post('/user');
});
```

Também podemos realizar a asserção verificando que o status da resposta também é 401:

```
it('Tenta cadastrar usuário estando logado mas sem possuir a role ADMIN',
async () => {
  const unauthorized = clientHTTP.authenticate(client, users[1]);
  const { response } = await unauthorized.post('/user');
  expect(response.status).toEqual(401);
});
```

Se quiser ter certeza (ou até realizar uma nova asserção), você pode verificar o valor de `response.data` e verá que nos dois testes as mensagens são diferentes e são, realmente, as mensagens que retornamos nos middlewares do Express.

Na sequência, vamos testar os cenários onde autenticamos a chamada como ADMIN. Sendo assim, precisamos validar dois casos:

- Tentar cadastrar usuário sem fornecer todos os parâmetros necessários e, consequentemente, não conseguindo cadastrar.

- Tentar cadastrar usuário fornecendo tudo corretamente e, consequentemente, conseguindo cadastrar.

Vamos para o primeiro caso e, com isso, criar mais um `it`:

```
it('Admin logado, tenta cadastrar usuário sem informar todos os dados',
async () => {
});
```

Precisaremos autenticar essa chamada como um usuário admin, então já vamos realizar essa operação:

```
it('Admin logado, tenta cadastrar usuário sem informar todos os dados',
async () => {
  // autenticamos como admin
  const authorized = clientHTTP.authenticate(client, admin);
});
```

Agora, precisamos criar um usuário sem ter todas as informações. Podemos utilizar a função `createUser`, exportada pelas nossas factories em `utils/create`, para nos ajudar. No começo do arquivo, vamos adicioná-la às importações:

```
// adicionamos o import de createUser
import { createUserListWithAdmin, clientHTTP, createUser } from
'utils/create';
```

E então, no teste, vamos chamar essa função extraindo apenas alguns valores, como `email` e `password`. Vamos aproveitar e já criar uma variável chamada `newUser` com essas informações também:

```
it('Admin logado, tenta cadastrar usuário sem informar todos os dados',
async () => {
  const authorized = clientHTTP.authenticate(client, admin);
  // executamos createUser e extraímos
  // somente email e password
  const { email, password } = createUser();
  // criamos um objeto novo com o mesmo email e password
  const newUser = { email, password };
});
```

Já podemos disparar nossa requisição do tipo `POST` informando os dados desse novo usuário. Dessa vez, o `status` de retorno deverá ser o número 400, já que estamos enviando uma requisição que não possui os campos necessários:

```
it('Admin logado, tenta cadastrar usuário sem informar todos os dados',
async () => {
  const authorized = clientHTTP.authenticate(client, admin);
  const { email, password } = createUser();
  const newUser = { email, password };
  // realizamos a requisição passando os dados do novo usuário
  const { response } = await authorized.post('/user', newUser);
  // asseguramos que o status recebido foi 400
  expect(response.status).toEqual(400);
});
```

Caso tenha curiosidade e queira ver o valor de `response.data`, você verá que a mensagem dos campos está retornando corretamente.

Agora vamos para o cenário em que um usuário é criado com sucesso. Vamos criar mais um `it` e já autenticar o cliente HTTP:

```
// novo it com título
it('Admin logado, cria usuário com sucesso, informando todos os dados',
async () => {
  // cliente HTTP autenticado
  const authorized = clientHTTP.authenticate(client, admin);
});
```

Para criar um usuário novo, podemos chamar a função `createUser`. No entanto, essa função também gera um `uid`. Como esse campo também será gerado automaticamente ao salvar o usuário em uma base, podemos ignorá-lo. Vamos utilizar desestruturação com o operador rest para pegar somente os demais campos:

```
it('Admin logado, cria usuário com sucesso, informando todos os dados',
async () => {
  const authorized = clientHTTP.authenticate(client, admin);
  // executamos a função createUser
  // separados o uid dos demais campos
```

```
  // criando a variável newUser sem o uid
  const { uid, ...newUser } = createUser();
});
```

Vamos disparar nossa requisição novamente com esses dados:

```
it('Admin logado, cria usuário com sucesso, informando todos os dados',
async () => {
  const authorized = clientHTTP.authenticate(client, admin);
  const { uid, ...newUser } = createUser();
  // disparamos a requisição com os dados do usuário
  const request = await authorized.post('/user', newUser);
});
```

E então podemos criar nossas asserções. Ao criar um usuário com sucesso, o status que receberemos é 201:

```
it('Admin logado, cria usuário com sucesso, informando todos os dados',
async () => {
  const authorized = clientHTTP.authenticate(client, admin);
  const { uid, ...newUser } = createUser();

  const request = await authorized.post('/user', newUser);

  expect(request.status).toEqual(201);
});
```

Também receberemos como resposta (no campo `data`) os dados do usuário criado. Já que o campo `uid` é criado nesse objeto a ser salvo na base de dados, podemos removê-lo utilizando desestruturação e informar somente os dados restantes que foram criados.

Após isso, vamos utilizar a função `toMatchObject` para verificar se os dados de `response.data` são iguais aos do objeto `newUser`:

```
it('Admin logado, cria usuário com sucesso, informando todos os dados',
async () => {
  const authorized = clientHTTP.authenticate(client, admin);
  const { uid, ...newUser } = createUser();
```

```
  const request = await authorized.post('/user', newUser);

  expect(request.status).toEqual(201);
  expect(request.data).toMatchObject(newUser);
});
```

Dessa forma, mesmo que `request.data` tenha informações extras (como `uid`), garantimos que os demais campos são iguais aos valores de `newUser`.

Pronto, temos mais um teste feito! Com isso, ficam faltando somente os métodos de remoção (`DELETE`) e atualização (`PATCH`) dessa rota de usuário. O endpoint de `/users`, que lista todos os dados de todos os usuários, também só possui um método e é exatamente igual ao que acabamos de fazer: é só realizar uma autenticação com qualquer usuário (nem precisa ser admin) e verificar os dados de resposta.

Deixo esses dois métodos e o endpoint restante de usuários como um desafio desta etapa para você, lembrando que no GitHub temos todos esses testes prontos, caso precise dar uma olhada, ok?

A seguir, vamos testar mais um endpoint: o de login.

9.3 Testando endpoint de login

Este endpoint também será bem simples. Vamos criar um arquivo `auth.route.integration.js` na pasta `__tests__` e copiar todo o conteúdo que criamos no teste anterior, desde os imports até os hooks, deixando de lado somente os testes:

```
// arquivo __tests__/auth.route.integration.js

// importamos app
import app from '../src/index';
```

```
// importamos factories
import { createUserListWithAdmin, clientHTTP, createUser } from
'utils/create';

// realizamos os mocks
import { loadDatabase } from '@jsassertivo/cli/src/database/file.js';
jest.mock('@jsassertivo/cli/src/database/file.js');
jest.mock('@jsassertivo/cli/src/utils/logger.js');

const users = createUserListWithAdmin();
const [admin] = users;
loadDatabase.mockResolvedValue(users);

// mesmos hooks e variáveis
let server;
beforeAll(() => {
  server = app();
});

let client;
beforeEach(() => {
  client = clientHTTP.create(server);
});

afterAll(() => {
  server.close();
});
```

Vamos criar um `describe` para a rota `/auth/login` e um outro para o método `POST`:

```
describe('/auth/login', () => {
  describe('POST', () => {
  });
});
```

Agora, vamos começar pelo cenário em que tentamos realizar o login de um usuário inexistente. Dentro desses `describes`, vamos criar um `it`:

```
it('Não consegue logar usuário com dados inválidos', async () => {
});
```

Como precisaremos simular um usuário que não está na base, vamos usar a função `createUser` para nos ajudar com isso. Vamos executá-la e extrair os campos `userName` (aproveitando para renomeá-lo para `username` com `n` minúsculo) e `password`. Após isso, vamos criar uma variável `user` com esses campos:

```
it('Não consegue logar usuário com dados inválidos', async () => {
  // executamos a função createUser
  // e pegamos os valores de username e password
  const { userName: username, password } = createUser();
  // criamos um objeto user com os dados de username e password
  const user = { username, password };
});
```

Só precisamos disparar nossa requisição para o endpoint `/auth/login` informando esse usuário:

```
it('Não consegue logar usuário com dados inválidos', async () => {
  const { userName: username, password } = createUser();
  const user = { username, password };
  // disparamos a requisição
  const { response } = await client.post('/auth/login', user);
});
```

E verificar se o status da resposta foi 404, já que os dados do usuário informado não correspondem a nenhum usuário do sistema:

```
it('Não consegue logar usuário com dados inválidos', async () => {
  const { userName: username, password } = createUser();
  const user = { username, password };
  const { response } = await client.post('/auth/login', user);
  // asserção que verifica o status 404
  expect(response.status).toEqual(404);
});
```

Com isso, nosso teste já valida a tentativa de login de um usuário inexistente.

Agora vamos para o cenário em que um usuário consegue realizar o login. Vamos criar um novo `it` dentro desse mesmo `describe`:

```
it('Realiza o login de um usuário com dados válidos', async () => {
});
```

Vamos criar uma variável `user` com os dados de algum usuário válido, como, por exemplo, o `admin`:

```
it('Realiza o login de um usuário com dados válidos', async () => {
    const user = {
        username: admin.userName,
        password: admin.password
    };
});
```

Com isso, é só dispararmos a requisição e verificar se o `status` retornado é 200. Caso queira, você também pode validar se o campo `data` é igual a um objeto contendo os dados do usuário:

```
 it('Realiza o login de um usuário com dados válidos', async () => {
    const user = {
      username: admin.userName,
      password: admin.password
    };
    // disparamos a requisição
    const request = await client.post('/auth/login', user);

    // verificamos o status 200
    expect(request.status).toEqual(200);
    // opcional: verificamos se o campo request.data
    // é igual a um objeto contendo o uid do usuário logado
    expect(admin).toEqual(expect.objectContaining(request.data));
});
```

Também é interessante garantirmos que esse dado foi inserido via `cookie` na requisição. Todos os cookies inseridos na requisição ficam disponíveis em `request.headers`, que é um objeto cuja chave `set-cookie` possui os cookies que devem ser inseridos.

Devemos acessá-lo através de `request.headers['set-cookie']`. Isso nos retornará um array com os valores de cookie a serem inseridos. Utilizando a desestruturação, podemos pegar o primeiro valor, que é o que nos interessa:

```
it('Realiza o login de um usuário com dados válidos', async () => {
    const user = {
        username: admin.userName,
        password: admin.password
    };
    const request = await client.post('/auth/login', user);
    // pegamos o primeiro cookie de request.headers['set-cookie']
    // através de desestruturação do array de cookies
    const [cookie] = request.headers['set-cookie'];

    expect(request.status).toEqual(200);
    expect(admin).toEqual(expect.objectContaining(request.data));
});
```

Com isso, podemos fazer uma asserção validando que o valor desse cookie é a string que contém o `uid` do usuário enviado na requisição:

```
it('Realiza o login de um usuário com dados válidos', async () => {
    const user = {
        username: admin.userName,
        password: admin.password
    };
    const request = await client.post('/auth/login', user);
    const [cookie] = request.headers['set-cookie'];

    expect(request.status).toEqual(200);
    expect(admin).toEqual(expect.objectContaining(request.data));
    // inserimos a asserção verificando que cookie
    // é uma string contendo o uid do admin
    expect(cookie).toEqual(expect.stringContaining(admin.uid));
});
```

Com isso, nossa rota de autenticação também já está testada nessa etapa de integração!

Não se esqueça de realizar aqueles últimos desafios, criando os testes de integração do endpoint `/users` e dos métodos `PATCH` e `DELETE` do endpoint `/user`. Com tudo o que vimos até agora tenho certeza de que você conseguirá sem problemas.

9.4 Próxima parada: testes de carga

Com isso, finalizamos mais um passo nessa jornada de criação de confiança no software que escrevemos.

Agora que já entendemos como os testes de integração funcionam, teremos os testes de carga como um conteúdo extra para garantir a qualidade da nossa API.

A ideia no próximo tópico é simular uma certa quantidade de usuários no nosso sistema e verificar como ele se comporta quando está sob uma certa demanda (ou "stress").

Vamos lá!

Capítulo 10
Testes de carga

Essa categoria de testes tem como objetivo identificar como uma aplicação se comporta ao ser acessada por determinada quantidade de usuários.

Geralmente, ferramentas de teste de carga também são usadas para testes de *stress*. A diferença é que, nos testes de carga, você tenta identificar como seu sistema funciona com X usuários acessando, já nos testes de stress, você tenta chegar ao limite do seu software para verificar quantos usuários simultaneamente ele conseguiria suportar. Conceitualmente eles são um pouco diferentes mas, na prática, acabam sendo bem parecidos.

Esta etapa ainda será realizada na API que testamos até o momento e vamos usar a ferramenta **Artillery** (https://artillery.io/) para esses testes.

10.1 Configurando o Artillery

Assim como as demais, essa ferramenta já está instalada no projeto e sua configuração é bem simples. Após estar instalada em um projeto (ou globalmente), basta criar um arquivo YAML qualquer para ser utilizado como base para a criação dos cenários de carga.

Caso você não tenha familiaridade com arquivos em formato `.yml`, eles são bem parecidos com JSON, com algumas peculiaridades, e são muito utilizados para configurações de serviços. Este site https://www.json2yaml.com/convert-yaml-to-json tem um exemplo bem claro da conversão de um arquivo `.json` para `.yml` e alguns segundos de estudo já são o suficiente para que você entenda tudo o que vamos precisar para a nossa configuração.

Agora vamos criar um arquivo chamado `load.yml` dentro da nossa pasta `__tests__` para realizarmos sua configuração. Após criar esse arquivo, precisaremos indicar duas seções: uma `config` e uma `scenarios`. Vamos deixá-las vazias por enquanto:

```
config:

scenarios:
```

Dentro de `config` podemos preencher algumas informações de configuração para nosso teste funcionar, como o campo `target`, que funciona de forma bem parecida com o `baseURL`, que criamos em nosso cliente de axios anteriormente. Vamos preencher essa informação com as informações básicas da nossa API:

```
config:
  # preenchemos o target com a URL da API
  target: http://localhost:8080/api

scenarios:
```

Além do campo `target`, precisamos indicar algumas fases de teste com o nome `phases`. É dentro dessas fases que indicamos a duração do teste e a quantidade de usuários virtuais que utilizarão nosso sistema nesse teste. Esses campos se chamam `duration` (duração) e `arrivalRate` (taxa de chegada), respectivamente. Vamos indicar que nosso teste durará 15 segundos e simulará requisições com 50 usuários acessando ao mesmo tempo:

```
config:
  target: http://localhost:8080/api
  #  criamos o campo com fases
  phases:
    # indicamos que terá 15 segundos de duração
    - duration: 15
    # com 50 usuários simultâneos
      arrivalRate: 50

scenarios:
```

Dentro da opção `scenarios`, indicaremos quais cenários de testes vamos realizar. Com a opção `flow`, indicamos vários fluxos de teste. Dentro dela, podemos indicar o método `http` que deve ser utilizado na requisição e também uma URL. Vamos criar uma para nosso endpoint de `/auth/login`:

```
config:
  target: http://localhost:8080/api
  phases:
    - duration: 15
      arrivalRate: 50

scenarios:
  # criamos o flow
  - flow:
    # indicamos que realizaremos uma requisição POST
      - post:
        # indicamos a URL
        url: /auth/login
```

Agora, para que possamos efetuar um login nesse teste, só precisamos indicar as variáveis contendo `username` e `password`. Vamos colocá-las usando o campo `json`:

```
config:
  target: http://localhost:8080/api
  phases:
    - duration: 15
      arrivalRate: 50

scenarios:
  - flow:
    - post:
      url: /auth/login
      #  colocamos o campo json
      json:
        # preenchemos os valores de username e password
        username: admin
        password: admin
```

10.2 Executando os testes de carga

Para que consigamos realizar o teste, é necessário que o servidor esteja rodando. Portanto, é interessante abrir uma aba do terminal apenas para rodar o servidor e, em outra, disparar os testes.

Agora, tudo o que precisamos fazer é executar o comando `artillery` apontando para o caminho desse arquivo de configuração. Já existe um comando pré-configurado no `package.json` desse projeto: basta executar `npm run test:load` que o teste de carga vai iniciar.

Relatório dos testes

Após rodar o servidor da aplicação e os testes, você terá alguns relatórios no seu terminal, mais ou menos assim:

```
All virtual users finished
Summary report @ 17:59:58(-0300) 2020-11-12
  Scenarios launched:  750
  Scenarios completed: 750
  Requests completed:  750
  Mean response/sec: 48.51
  Response time (msec):
    min: 2.4
    max: 31.1
    median: 3.1
    p95: 3.8
    p99: 5.5
  Scenario counts:
    0: 750 (100%)
  Codes:
    200: 750
```

Dentro desse relatório, conseguimos ver como os testes se comportaram, quantos cenários e requisições foram disparadas e completadas. Nesse caso, foram 750, justamente porque 50 requisições foram enviadas a cada segundo, e tivemos 15 segundos.

Também conseguimos ver o valor mínimo, máximo, mediano, p95 e p99 (que são percentil, indicando pelo menos 95% e 99% dos cenários) e, ao final, temos que ver a quantidade total de cenários e os status que as requisições responderam junto de sua contagem.

É provável que você receba os logs de outros relatórios enquanto os testes são executados. Esse que coloquei como exemplo é o relatório gerado ao final, mas não se assuste caso ele apareça algumas vezes conforme os testes forem rodando.

Claro que todo esse relatório pode mudar de uma máquina para outra, então provavelmente o resultado que você vai obter em seu computador será diferente.

Existem várias opções e configurações para executar testes de carga em sua aplicação. A documentação do Artillery disponível em https://artillery.io/docs/ é bem completa e detalhada. Sinta-se à vontade para brincar com Artillery e criar mais cenários de testes agora que você já entendeu como sua estrutura principal funciona. Você pode aproveitar também para testar outros endpoints de nossa API.

10.3 Cansei de testar APIs, não vamos ter nada visual?

Com esta etapa, finalizamos as nossas camadas de testes em aplicações back-end.

Já parou para analisar tudo o que testamos até aqui? Aplicamos de forma coerente testes unitários, integrados e um exemplo de teste de carga em nossa API.

Agora chegou a hora de realizar testes em aplicações com uma outra perspectiva de desenvolvimento no ecossistema JavaScript: vamos testar uma aplicação front-end.

Espero que tenha gostado do que aprendeu até o momento, ainda temos muita coisa interessante para aprender nesta jornada!

Parte 4: Testando aplicações front-end

Os testes mudam um pouco quando falamos de aplicações que interagem com o navegador.

Agora está na hora de testarmos a camada de front-end da nossa aplicação, também realizando os testes de unidade e integração.

Depois, vamos aplicar testes de regressão visual nos nossos componentes para evitar que mudanças indesejadas na nossa interface aconteçam.

Capítulo 11
Testes unitários nos componentes da aplicação

Para começarmos a entender como os testes no front-end funcionam, primeiro precisamos conhecer um pacote chamado *JSDOM*, lembra dele? Comentamos sobre ele bem vagamente alguns capítulos atrás e chegou a hora de entendermos para que ele serve e como ele é fundamental para os testes que vamos escrever.

Você já deve ter ouvido falar no termo "especificação", certo? Basicamente, especificações são definições de regras, comportamentos e funcionamentos de um determinado sistema ou de alguma implementação. O JSDOM (https://github.com/jsdom/jsdom) é um pacote que tem como papel implementar algumas especificações da Web, como o DOM e o HTML. Sim! O DOM e o HTML são especificações (que você pode verificar através dos links https://dom.spec.whatwg.org/ e https://html.spec.whatwg.org/multipage/), e cada navegador implementa essas especificações de uma determinada maneira.

É o JSDOM que permite que os testes do lado do front-end se tornem realidade, pois é ele o responsável por implementar as diversas APIs e especificações de um navegador. O próprio Jest cuida de preparar um ambiente com o JSDOM para nossa utilização e por isso não manipulamos seus objetos diretamente, mas tivemos que ajustar uma configuração anteriormente.

Agora que já sabemos como algumas das coisas dos testes que vamos realizar funcionam "por baixo dos panos", vamos começar a testar alguns componentes da nossa interface.

Um pequeno aviso sobre as tecnologias dessa parte

Essa aplicação front-end foi desenvolvida utilizando **React** (https://pt-br.reactjs.org/), uma biblioteca para criação de interfaces que tem tomado cada vez mais espaço na comunidade e nas empresas quando o assunto é desenvolvimento de UI (*user interface*, ou interface de usuários).

Pode ser que você tenha mais contato com aplicações escritas utilizando Angular, Vue ou até mesmo que não utilize nenhuma dessas ferramentas. Não se preocupe!

Embora as implementações e os componentes que vamos testar sigam os padrões do React, a biblioteca de testes que utilizaremos se chama *Testing Library* (https://testing-library.com/). Ela pode ser aplicada nos testes de todos os frameworks mais famosos do mercado e inclusive em aplicações que não utilizam qualquer framework!

Ela também aplica um mesmo padrão de consulta de elementos em tela e até de manuseio de eventos, além de encorajar diversas boas práticas de escrita nos componentes, principalmente relacionadas à acessibilidade.

Então, por mais que você não utilize especificamente o React, com certeza o conteúdo que veremos nos próximos capítulos poderá ser

reaproveitado perfeitamente para as necessidades de qualquer aplicação que você for escrever.

Além de utilizar React, a aplicação utiliza Redux (https://redux.js.org/) e React-Redux (https://react-redux.js.org/) para gerenciamento de estado e Styled Components (https://styled-components.com/) como solução de estilo. Embora seja uma aplicação pequena e não tenha necessidade de tanta complexidade, a ideia é poder deixar os testes o mais próximo possível das aplicações que você pode encontrar no mercado.

11.1 Estrutura do projeto, instalação e configuração

O código-fonte que testaremos pode ser acessado no mesmo repositório em que temos trabalhado até o momento (https://javascriptassertivo.com.br/) e se encontra na pasta `projetos/04-testando-aplicacoes-front-end`.

Assim como os projetos anteriores, você pode apagar a pasta `__tests__` para que possamos escrever os testes juntos, a única diferença é que dessa vez ela está dentro da pasta `src` e vamos fazer nossos testes por lá. Essa mudança foi necessária para facilitar algumas configurações iniciais que já vieram por padrão ao criar o projeto com `create-react-app` (*boilerplate* de configuração do React para desenvolvimento). Também existe um arquivo `README.md` na pasta raiz, que possui mais alguns detalhes sobre como o projeto foi construído e sua estrutura.

Vamos dar uma olhada na estrutura principal. Dentro da pasta `src` encontra-se o código-fonte da aplicação dividido da seguinte maneira:

- `clients` : clientes intermediários que ligam os componentes às APIs externas, como:
  - `http` : para a realização de requisições HTTP;
  - `storage` : para a manipulação de dados de `cookie/localStorage` .
- `components` : componentes mais básicos da aplicação que servem somente de interação de usuário e não manipulam nenhum estado global. Cada componente está em uma pasta separada, que contém:
  - `index.js` : arquivo do componente em si;
  - `styles.js` : arquivo com os estilos do componente;
  - `stories.js` : arquivo de configuração do storybook do componente.
- `hooks` : alguns hooks do React reutilizáveis;
- `mocks` : alguns dados para utilização nos arquivos;
- `pages` : componentes principais das páginas (`login` e `dashboard`), que montam as telas da aplicação utilizando os componentes da pasta `components` e também utilizam dados de estado definidos globalmente;
- `providers` : os provedores de informação (e Providers React) da aplicação, divididos em:
  - `notification` : provê a configuração para a utilização das notificações de sucesso/erro;
  - `redux` : provê a configuração para a utilização do Redux como gerenciamento global de estado;
  - `theme` : provê o tema da aplicação para reúso dos componentes.
- `routes` : configurações para utilização das rotas públicas e privadas;
- `store` : arquivos relacionados ao estado global da aplicação, contém as configurações principais nos arquivos `index.js` , as configurações de middlewares em `middlewares.js` e a composição dos reducers em `reducers.js` . Também possui as subpastas, cada uma com suas respectivas `actions` , `reducers` e `selectors` , sendo:
  - `notification` : relacionada a notificações de sucesso/erro;

    - `profiles` : relacionada aos dados de perfis de usuários;
    - `user` : relacionada aos dados do usuário autenticado.
- `styles` : possui os estilos globais da aplicação no arquivo `reset.js` e o tema no arquivo `theme.js` ;
- `app.js` : inicia as aplicações com todos os providers, rotas e aplica o reset de estilos;
- `index.js` : instancia a aplicação utilizando ReactDOM.

Existem alguns outros arquivos que não vale a pena comentarmos aqui. Com tudo isso, já podemos ter uma ideia de tudo o que testaremos ao longo dos próximos capítulos!

O processo de instalação e execução é o mesmo. Basta clonar o repositório, acessar a pasta do projeto e instalar as dependências com `npm i .`

Você pode executar `npm start` para rodar a aplicação (somente o front-end), `npm run storybook` para olhar o storybook (um catálogo/playground com os componentes básicos de interface), e também `npm run start:api` para rodar a API do projeto anterior (back-end). Isso será necessário para que o projeto funcione corretamente, já que essa aplicação se conecta à API que utilizamos anteriormente. Existe também um script extra, o `npm run start:all` , que inicia o servidor de desenvolvimento da aplicação front-end e também inicia a API do back-end com um só comando. Caso prefira, você pode utilizá-lo diretamente.

Pegue alguns minutos para executar a aplicação e o storybook para se ambientar ao projeto que testaremos.

Vamos começar essa etapa testando alguns componentes puramente visuais, que não se integram a nenhum fator exterior (como um estado global) e são responsáveis apenas por renderizar algum conteúdo em tela.

11.2 Testando componentes puramente visuais

Iniciando por um dos componentes mais básicos: o botão

Quando pensamos em interface e interação com usuários através de telas em um sistema, a primeira coisa que (geralmente) vem à nossa mente é um botão: o componente mais simples e utilizado em grande parte dos sistemas que encontramos no mercado.

Justamente por ser um componente tão útil é que vamos começar nossos testes com ele. Geralmente componentes de botão possuem muito mais características visuais do que lógicas, sendo responsáveis apenas por renderizar botões com alguns estilos predefinidos e, simplesmente, executando uma ação de `click` fornecido via algum parâmetro (ou `prop`, no caso do React).

Vamos seguir a mesma estrutura dos testes unitários e de integração que fizemos no capítulo anterior, portanto, dentro da pasta `src/__tests__` crie uma pasta `components` e, dentro dela, o arquivo `button.unit.js`. É nesse arquivo que realizaremos o teste unitário do nosso botão.

Dentro desse arquivo, vamos importar a função `render`, exportada pelo pacote `@testing-library/react`, que nos permitirá renderizar o nosso componente:

```
// arquivo src/__tests__/components/button.unit.js
// importamos a função render
import { render } from '@testing-library/react';
```

Vamos importar também o componente `Button` para que possamos testar:

```
import { render } from '@testing-library/react';

// importamos o botão
import Button from '../../components/button';
```

Agora vamos criar nosso bloco `describe`. Podemos dar o nome do nosso próprio componente para ele. Vamos aproveitar e criar nosso primeiro `it` com um texto indicando que a renderização básica do botão será testada:

```
import { render } from '@testing-library/react';

import Button from '../../components/button';

// criamos describe/it
describe('<Button />', () => {
  it('Renderiza um botão corretamente', () => {
  });
});
```

Para que possamos renderizar esse botão, basta chamarmos a função `render` executando nosso componente.

Para executar um componente React, basta colocá-lo entre os caracteres que abrem e fecham uma tag do HTML, como `<Button />`. Caso esse componente receba algum conteúdo como filho (`children`), ele pode ser passado dentro da tag como `<Button>conteudo</Button>` e, caso precise passar algum dado extra (como o tipo do botão, por exemplo), você pode passar via `props`, que são iguais aos atributos HTML.

Por exemplo, para mudar a cor do botão para azul, bastaria executá-lo como `<Button type="blue">conteudo</Button>`, pois `type` é uma propriedade mapeada dentro do componente na aplicação que muda sua cor para algumas variantes disponíveis. É importante saber que componentes React são escritos em JSX e precisam começar com a letra maiúscula. Ou seja, caso executássemos o botão como `<button>conteudo</button>`, criaríamos uma tag `button` normal e não utilizaríamos o componente que criamos.

Vamos lá:

```
import { render } from '@testing-library/react';
```

```
import Button from '../../components/button';

describe('<Button />', () => {
  it('Renderiza um botão corretamente', () => {
    // renderizamos o botão com a função render
    render(<Button>conteudo</Button>);
  });
});
```

Em nosso teste, o botão será renderizado em uma cópia do DOM feita pelo JSDOM. Tudo o que precisamos fazer é encontrar esse elemento e verificar se ele, de fato, foi renderizado. Afinal, poderíamos ter alguma lógica interna ao componente, que não retornasse nada em sua renderização e não exibisse nenhum resultado na tela. Para consultar o elemento, existem algumas formas diferentes, a mais indicada é importar um utilitário chamado `screen`, da própria *Testing Library*, que permite a consulta de alguns elementos por seus textos e roles de acessibilidade na tela. Vamos importá-lo:

```
// importamos screen logo após render
import { render, screen } from '@testing-library/react';
```

Já podemos realizar algumas consultas em tela. Existem várias formas de consultar um elemento e elas podem variar principalmente quando existe algum cenário onde não retornamos nada e queremos garantir que nada foi renderizado na tela. No link https://testing-library.com/docs/dom-testing-library/api-queries/, está a documentação completa das consultas (`queries`) que podem ser realizadas, mas elas podem ser resumidas da seguinte forma:

- Queries começadas com `getBy*` : retornam o primeiro elemento encontrado com o seletor informado e disparam um erro caso não encontrem nada. Caso você precise encontrar vários elementos com um mesmo seletor, utilize a query `getAllBy*` , que retorna uma lista de elementos de um mesmo seletor.
- Queries começadas com `queryBy*` : retornam o primeiro elemento encontrado com um seletor informado e retornam

`null` caso nenhum elemento exista. Caso você precise encontrar vários elementos com um mesmo seletor, utilize a query `queryAllBy*`, que retorna uma lista de elementos de um mesmo seletor.

- Queries começadas com `findBy*`: retornam uma `Promise`, que é resolvida com o primeiro elemento encontrado com um seletor informado, e a `Promise` é rejeitada caso nenhum elemento exista. Caso você precise encontrar vários elementos com um mesmo seletor, utilize a query `findAllBy*`, que retorna uma lista de elementos de um mesmo seletor.

Isso quer dizer que as consultas que utilizaremos serão feitas em cima dessas três versões. Cada uma delas pode ser dividida em alguns casos como:

text: procura um elemento por texto, resultando nas funções `getByText`, `getAllByText`, `queryByText`, `queryAllByText`, `findByText` e `findAllByText`;

role: procura um elemento por sua role de acessibilidade na tela, resultando nas funções `getByRole`, `getAllByRole`, `queryByRole`, `queryAllByRole`, `findByRole` e `findAllByRole`;

placeholder text: procura um elemento pelo seu texto de placeholder (inputs), resultando nas funções `getByPlaceholderText`, `getAllByPlaceholderText`, `queryByPlaceholderText`, `queryAllByPlaceholderText`, `findByPlaceholderText` e `findAllByPlaceholderText`;

Também podem ser consultados elementos por `label text`, `alt text`, `title`, `display value` e `test id`, seguindo a mesma lógica de consulta e das funções.

No nosso caso podemos, por exemplo, verificar se um elemento com o texto `conteudo` foi inserido na tela através da consulta por texto. Podemos fazer isso com a função `screen.getByText`:

```
import { render, screen } from '@testing-library/react';

import Button from '../../components/button';

describe('<Button />', () => {
  it('Renderiza um botão corretamente', () => {
    render(<Button>conteudo</Button>);
    // realizamos a consulta com getByText passando o texto do botão
    const button = screen.getByText('conteudo');
  });
});
```

Ao salvar o arquivo e rodar os testes (com `npm test` ou `npm run test:watch`), teremos um erro nos arquivos de estilo do nosso componente. Isso ocorre porque esse componente está atrelado a um tema utilizando os provedores (`providers`) do React. Quando tentamos renderizá-lo sem realizar nenhuma configuração desse tema, recebemos um erro. Poderíamos resolver isso importando e configurando esse provedor do tema diretamente no nosso teste, mas teríamos que fazer isso para praticamente todos os componentes.

Já deixei separada uma função que realiza esse trabalho para nós. Vamos importar a função `renderWithTheme` do arquivo `src/testUtils.js` e utilizá-la em vez da função `render` da biblioteca RTL (*React Testing Library*):

```
// apagamos o import da função render
import { screen } from '@testing-library/react';
// importamos a função renderWithTheme do arquivo testUtils
import { renderWithTheme } from '../../testUtils';

import Button from '../../components/button';

describe('<Button />', () => {
  it('Renderiza um botão corretamente', () => {
    // ajustamos nosso teste para usar renderWithTheme
    renderWithTheme(<Button>conteudo</Button>);
    const button = screen.getByText('conteudo');
```

```
  });
});
```

Com isso, nossos testes passam, mas ainda não existe nenhuma asserção válida. O arquivo `src/setupTests.js` importa um conteúdo de `@testing-library/jest-dom`, uma biblioteca que contém asserções customizadas da Testing Library. Podemos verificar se um elemento foi renderizado através da asserção `.toBeInTheDocument`, da seguinte forma:

```
import { screen } from '@testing-library/react';
import { renderWithTheme } from '../../testUtils';

import Button from '../../components/button';

describe('<Button />', () => {
  it('Renderiza um botão corretamente', () => {
    renderWithTheme(<Button>conteudo</Button>);
    const button = screen.getByText('conteudo');
    // inserimos asserção
    expect(button).toBeInTheDocument();
  });
});
```

Dessa forma, garantimos que o elemento foi renderizado na tela corretamente. Se quisermos deixar registrada a estrutura renderizada do nosso componente, podemos utilizar a combinação com `toMatchSnapshot`. Essa função captura o resultado de algum conteúdo em texto (no caso, a árvore DOM renderizada em nosso componente) para ser comparado a cada vez que os testes rodam. Vamos adicioná-la:

```
import { screen } from '@testing-library/react';
import { renderWithTheme } from '../../testUtils';

import Button from '../../components/button';

describe('<Button />', () => {
  it('Renderiza um botão corretamente', () => {
    renderWithTheme(<Button>conteudo</Button>);
```

```
    const button = screen.getByText('conteudo');
    expect(button).toBeInTheDocument();
    // inserimos snapshot
    expect(button).toMatchSnapshot();
  });
});
```

Com isso, teremos um log no nosso terminal, indicando que o snapshot foi criado:

```
 › 1 snapshot written.
Snapshot Summary
 › 1 snapshot written from 1 test suite.
```

E você também pode ver o resultado desse snapshot na pasta `__snapshots__` dentro da pasta do teste que estamos escrevendo. Ao dar uma olhada lá, podemos ver o seguinte:

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`<Button /> Renderiza um botão corretamente 1`] = `
<span>
  conteudo
</span>
`;
```

Esse arquivo contém exatamente a marcação do DOM que foi renderizada pelo nosso componente. Estranho que não apareceu o botão, não é?

Isso acontece pois estamos consultando nosso elemento diretamente pelo seu texto e, dentro do botão, existe uma tag `span` também. Vamos trocar nossa consulta para `getByRole('button')` em vez de `getByText('conteudo')` e ver o que acontece:

```
import { screen } from '@testing-library/react';
import { renderWithTheme } from '../../testUtils';

import Button from '../../components/button';

describe('<Button />', () => {
```

```
  it('Renderiza um botão corretamente', () => {
    renderWithTheme(<Button>conteudo</Button>);
    // alteramos a consulta para role
    const button = screen.getByRole('button');
    expect(button).toBeInTheDocument();
    expect(button).toMatchSnapshot();
  });
});
```

Com isso, já temos um erro no nosso terminal:

```
- Snapshot  - 0
+ Received  + 5

+ <button
+   class="sc-bdfBwQ iOlsKH"
+   type=""
+ >
    <span>
        conteudo
    </span>
+ </button>
```

Esse erro indica a diferença do nosso snapshot: inicialmente recebemos somente um span com nosso texto e, após esse ajuste (consultando agora pela "role", que é o "papel" que o botão exerce em uma tela), recebemos o elemento como esperávamos!

Para atualizar um snapshot que quebrou, basta executar a opção `-u` ao rodar os testes. Caso esteja executando com `watch`, no comando `npm run test:watch`, você pode teclar `w` para escolher mais opções e depois teclar `u`. Com isso, o snapshot será atualizado:

```
 › 1 snapshot updated.
Snapshot Summary
 › 1 snapshot updated from 1 test suite.
```

E nosso teste inicial passa corretamente. É interessante utilizar as consultas dos elementos de forma correta principalmente quando o assunto é acessibilidade. Consultas que utilizam `role` do elemento

ajudam a garantir que o elemento está assumindo um papel correto na tela que foi renderizada, auxiliando em questões de acessibilidade e leitores de tela.

Só snapshots são o suficiente?

É uma dúvida que sempre surge quando o assunto é testar componentes de interface e, na minha opinião, a resposta é simples: **não**, só testes de snapshot não garantem a qualidade necessária dos nossos componentes.

Ao gerar um report da cobertura de testes do nosso componente, veremos que ele está com a cobertura bem próxima de 100%, mesmo nosso teste tendo renderizado-o de forma bem simples. Isso acontece pela forma como o Jest contabiliza as linhas executadas. No entanto, ainda existe uma parte crucial que ainda não verificamos no nosso componente: as ações de click. Afinal, um botão deve executar alguma coisa ao receber um evento, certo?

Sem contar que, como acabamos de ver, testes de snapshot são facilmente atualizados e necessitam de um olhar humano para que alguém veja com cautela a renderização de um componente. É normal que, no dia a dia, nem todos possuam esse cuidado, sendo comum haver testes de snapshot que simplesmente são atualizados sem que a pessoa que está executando os testes entenda o conteúdo que foi modificado.

Por isso, snapshots podem ser seus aliados nos testes, mas ter somente eles tornará suas asserções muito frágeis. Então, vamos evitar criar testes só de snapshots, pois já sabemos que eles não garantem muita coisa, combinado?

Vamos voltar ao nosso teste e garantir que, ao executarmos um click, nosso botão vai disparar uma função. Vamos criar um novo bloco de `it`:

```
import { screen } from '@testing-library/react';
import { renderWithTheme } from '../../testUtils';
```

```
import Button from '../../components/button';

describe('<Button />', () => {
  it('Renderiza um botão corretamente', () => {
    renderWithTheme(<Button>conteudo</Button>);
    const button = screen.getByRole('button');
    expect(button).toBeInTheDocument();
    expect(button).toMatchSnapshot();
  });

  // criamos novo it
  it('Executa ações de click ao receber a função por prop', () => {
  });
});
```

Faremos o mesmo processo de renderizar nosso botão com o tema, no entanto, desta vez passaremos uma função de click para ele. É possível atribuir funções a eventos no React usando os atributos como da forma clássica no HTML. No HTML, embora não seja a melhor maneira, poderíamos usar o atributo `onclick` ; no React, utilizamos `onClick` (com o `C` maiúsculo, padrão camelCase). Existem vários eventos que podemos utilizar em nossos componentes, mas no nosso caso precisamos exatamente do `onClick` .

Para passarmos alguma função como `onClick` do nosso botão, podemos criar o nosso bom e velho mock do Jest através de `jest.fn()` . Vamos criar e passá-lo como propriedade do nosso componente. Vamos aproveitar e já consultar nosso elemento como fizemos anteriormente:

```
it('Executa ações de click ao receber a função por prop', () => {
    // criamos a função onClick
    const onClick = jest.fn();
    // e renderizamos o componente
    // fornecendo a função de click
    renderWithTheme(<Button onClick={onClick}>conteudo</Button>);
    // consultamos o botão renderizado
```

```
    const button = screen.getByRole('button');
});
```

Agora só precisamos disparar um evento de click nesse botão. Podemos fazer isso através do utilitário `fireEvent`, disponível no próprio pacote da `@testing-library/react`, ou importando um outro pacote chamado `@testing-library/user-event`, que também já está instalado. Ambos funcionam perfeitamente, a questão é que a biblioteca `user-event` simula alguns eventos de forma mais completa, já que um evento no DOM pode ser composto por vários outros eventos relacionados e também pode ser disparado junto a esses outros eventos. Ela realiza esse agrupamento de forma mais uniforme a um usuário interagindo com nosso componente.

Dito isso, vamos importá-la no começo do nosso arquivo:

```
import { screen } from '@testing-library/react';
// importamos a biblioteca userEvent
import userEvent from '@testing-library/user-event';
import { renderWithTheme } from '../../testUtils';

import Button from '../../components/button';
```

Podemos clicar no botão através da função `userEvent.click` e verificar se nossa função foi chamada corretamente:

```
it('Executa ações de click ao receber a função por prop', () => {
  const onClick = jest.fn();
  renderWithTheme(<Button onClick={onClick}>conteudo</Button>);
  const button = screen.getByRole('button');
  // executamos o click no botão
  userEvent.click(button);
  // realizamos a asserção verificando a chamada da função
  // onClick uma única vez
  expect(onClick).toHaveBeenCalledTimes(1);
});
```

Com isso, já verificamos nossa ação de click.

O botão também pode receber uma propriedade `icon`, que pode ser qualquer outro componente. Também é algo interessante de testarmos. Vamos criar um novo `it`:

```
it('Pode renderizar um ícone', () => {
});
```

Agora podemos criar qualquer elemento e passar na propriedade `icon`:

```
it('Pode renderizar um ícone', () => {
  const icon = <span>icone</span>;
  renderWithTheme(<Button icon={icon}>conteudo</Button>);
});
```

E verificar que o ícone foi renderizado corretamente:

```
it('Pode renderizar um ícone', () => {
  const icon = <span>ícone</span>;
  renderWithTheme(<Button icon={icon}>conteudo</Button>);
  // consultamos o ícone
  const icone = screen.getByText('icone');
  // verificamos se está no documento
  expect(icone).toBeInTheDocument();
});
```

O componente de input e suas variações

Vamos testar um outro componente bem importante na nossa aplicação, o de input. Esse componente se divide em outros três, de forma que, ao passar um `type`, ele renderize um campo de texto, de senha ou um select com várias opções.

Ainda sobre isso, o campo do tipo senha possui um botão, que deixa a senha visível ou não (alterando seu type temporariamente para `text`), então também precisamos garantir esse comportamento.

Vamos começar com o input de texto mais básico. Vamos criar o arquivo `input.unit.js` dentro da pasta `src/__tests__/components`. Já

vamos importar os módulos que vamos precisar, como o `renderWithTheme`, `screen` e `userEvent`, assim como o próprio componente:

```
// arquivo src/__tests__/components/input.unit.js
// importamos screen
import { screen } from '@testing-library/react';
// importamos userEvent
import userEvent from '@testing-library/user-event';
// importamos o utilitário de renderização com o tema
import { renderWithTheme } from '../../testUtils';
// importamos o componente
import Input from '../../components/input';
```

Vamos criar nosso bloco de `describe` com o primeiro `it`, onde vamos verificar o funcionamento do campo do tipo texto:

```
describe('<Input />', () => {
  it('Renderiza um campo de texto que pode ser preenchido', () => {
  });
});
```

Agora, vamos renderizar nosso input do tipo texto! Vamos aproveitar e passar um placeholder para que possamos consultá-lo depois:

```
describe('<Input />', () => {
  it('Renderiza um campo de texto que pode ser preenchido', () => {
    // renderizamos o input do tipo texto
    // com um placeholder
    renderWithTheme(<Input type="text" placeholder="campo de texto" />);
  });
});
```

Precisamos consultar esse elemento e garantir que ele está na tela:

```
it('Renderiza um campo de texto que pode ser preenchido', () => {
  renderWithTheme(<Input type="text" placeholder="campo de texto" />);
  // selecionamos o elemento por seu placeholder
  const input = screen.getByPlaceholderText('campo de texto');
  // validamos se está na tela
```

```
  expect(input).toBeInTheDocument();
});
```

Também é interessante garantir que, ao digitar qualquer informação, o usuário conseguirá alterar o valor do campo na tela. Vamos fazer isso através do `userEvent.type`, onde podemos passar o elemento que queremos digitar e algum valor de texto:

```
it('Renderiza um campo de texto que pode ser preenchido', () => {
    renderWithTheme(<Input type="text" placeholder="campo de texto" />);
    const input = screen.getByPlaceholderText('campo de texto');

    expect(input).toBeInTheDocument();
    // adicionamos a linha que digita no elemento
    userEvent.type(input, 'texto digitado pelo usuário');
});
```

E, para nos assegurarmos de que o valor está no input, podemos usar `toHaveValue`:

```
it('Renderiza um campo de texto que pode ser preenchido', () => {
    renderWithTheme(<Input type="text" placeholder="campo de texto" />);
    const input = screen.getByPlaceholderText('campo de texto');

    expect(input).toBeInTheDocument();

    userEvent.type(input, 'texto digitado pelo usuário');
    // verificamos que o input possui o valor digitado
    expect(input).toHaveValue('texto digitado pelo usuário');
});
```

Perfeito! A seguir, vamos realizar o teste para o nosso campo de senha. Vamos criar um novo `it`:

```
it('Renderiza um campo de senha que pode ficar visível', () => {

});
```

Vamos fazer o mesmo processo, renderizando o input (dessa vez, `type` deve possuir o valor `password`) e consultá-lo por seu

placeholder. Vamos também já realizar a asserção que valida que ele está na tela:

```
it('Renderiza um campo de senha que pode ficar visível', () => {
    // renderizamos o input do tipo senha
    renderWithTheme(<Input type="password" placeholder="campo de senha"
/>);
    // consultamos o input por seu placeholder
    const input = screen.getByPlaceholderText('campo de senha');
    // verificamos que está na tela
    expect(input).toBeInTheDocument();
});
```

Podemos também verificar que é possível digitar nesse campo:

```
it('Renderiza um campo de senha que pode ficar visível', () => {
    renderWithTheme(<Input type="password" placeholder="campo de senha"
/>);

    const input = screen.getByPlaceholderText('campo de senha');
    expect(input).toBeInTheDocument();

    // digitamos um valor no campo
    userEvent.type(input, 'senha super secreta');
    // verificamos que o valor foi inserido
    expect(input).toHaveValue('senha super secreta');
});
```

Para que possamos testar o funcionamento do click no botão que deixa a senha visível, podemos validar o atributo `type` do input. Quando a senha não é visível, o atributo `type` é `password`. Podemos fazer isso através da asserção `toHaveAttribute`:

```
  it('Renderiza um campo de senha que pode ficar visível', () => {
    renderWithTheme(<Input type="password" placeholder="campo de senha"
/>);

    const input = screen.getByPlaceholderText('campo de senha');
    expect(input).toBeInTheDocument();

    userEvent.type(input, 'senha super secreta');
```

```
    expect(input).toHaveValue('senha super secreta');

    // verificamos o atributo type do input
    expect(input).toHaveAttribute('type', 'password');
  });
```

Para que o tipo do input seja alterado, precisamos disparar um evento de clique no botão que fica dentro desse mesmo input. Vamos consultá-lo e depois disparar um click:

```
  it('Renderiza um campo de senha que pode ficar visível', () => {
    renderWithTheme(<Input type="password" placeholder="campo de senha"
/>);

    const input = screen.getByPlaceholderText('campo de senha');
    expect(input).toBeInTheDocument();

    userEvent.type(input, 'senha super secreta');
    expect(input).toHaveValue('senha super secreta');

    expect(input).toHaveAttribute('type', 'password');
    // consultamos o botão por sua role
    const button = screen.getByRole('button');
    // disparamos um click
    userEvent.click(button);
  });
```

Podemos verificar se o `type` do input é do tipo `text`:

```
  it('Renderiza um campo de senha que pode ficar visível', () => {
    renderWithTheme(<Input type="password" placeholder="campo de senha"
/>);

    const input = screen.getByPlaceholderText('campo de senha');
    expect(input).toBeInTheDocument();

    userEvent.type(input, 'senha super secreta');
    expect(input).toHaveValue('senha super secreta');

    expect(input).toHaveAttribute('type', 'password');
```

```
    const button = screen.getByRole('button');
    userEvent.click(button);
    // verificamos se o type agora é text
    expect(input).toHaveAttribute('type', 'text');
  });
```

Por último, mas não menos importante, vamos testar nosso componente de `select`. Semanticamente, uma tag `select` (e suas `options`) é muito diferente de uma tag de `input`, mas, para facilitar nossa aprendizagem, o projeto utiliza um mesmo componente e varia apenas seu tipo.

Nesse cenário, vamos precisar passar via `prop` um valor para `options`, para que o `select` possa ter algumas opções de valores. Essas `options` devem ser um array de objetos e cada objeto deve possuir a chave `text`, com o texto que será exibido, e `value`, com o valor da opção propriamente dita.

Vamos criar nosso `it` e renderizar esse elemento. Vamos colocar nossas opções em uma variável para facilitar nossa leitura:

```
// criamos nosso it
it('Renderiza um campo select com opções selecionáveis', () => {
    // criamos um array de opções com objetos com text/value
    const options = [
        { text: 'Administrador', value: 'ADMIN' },
        { text: 'Usuário', value: 'USER' },
    ];
});
```

Vamos renderizar nosso `input` do tipo `select` fornecendo essas opções e um placeholder como fizemos anteriormente:

```
it('Renderiza um campo select com opções selecionáveis', () => {
    const options = [
        { text: 'Administrador', value: 'ADMIN' },
        { text: 'Usuário', value: 'USER' },
    ];
    // renderizamos o input do tipo select com as opções
    // e o placeholder
```

```
    renderWithTheme(<Input type="select" options={options}
placeholder="selecione" />);
});
```

E vamos verificar que esse `select` foi renderizado corretamente:

```
it('Renderiza um campo select com opções selecionáveis', () => {
    const options = [
        { text: 'Administrador', value: 'ADMIN' },
        { text: 'Usuário', value: 'USER' },
    ];
    renderWithTheme(<Input type="select" options={options}
placeholder="selecione" />);
    // consultamos o select por seu placeholder
    const select = screen.getByPlaceholderText('selecione');
    // verificamos que esta na tela
    expect(select).toBeInTheDocument();
});
```

Na sequência, vamos verificar que podemos selecionar uma opção nesse `select`. Podemos fazer isso através da função `userEvent.selectOptions`, fornecendo o `select` como primeiro argumento e, como segundo, o texto do valor que queremos selecionar. Depois disso, podemos validar se o elemento possui o valor selecionado:

```
it('Renderiza um campo select com opções selecionáveis', () => {
    const options = [
        { text: 'Administrador', value: 'ADMIN' },
        { text: 'Usuário', value: 'USER' },
    ];
    renderWithTheme(<Input type="select" options={options}
placeholder="selecione" />);

    const select = screen.getByPlaceholderText('selecione');
    expect(select).toBeInTheDocument();

    // selecionamos a opção do select que possui o valor USER
    userEvent.selectOptions(select, 'USER');
    // verificamos se agora o valor é user
```

```
    expect(select).toHaveValue('USER');
});
```

Como em nossos testes a ideia é simular o comportamento de um usuário, existe uma maneira que eu, particularmente, prefiro para realizar a seleção dessas opções. Essa forma seria, em vez de fornecer diretamente um valor como segundo argumento da função `selectOptions`, fornecer um elemento que possui o texto daquela opção. Afinal, uma pessoa acessando o sistema não sabe ao certo o valor que a opção tem dentro de um `select`, ela apenas seleciona algo baseado em seu texto.

Vamos fazer esse ajuste:

```
it('Renderiza um campo select com opções selecionáveis', () => {
    const options = [
      { text: 'Administrador', value: 'ADMIN' },
      { text: 'Usuário', value: 'USER' },
    ];
    renderWithTheme(<Input type="select" options={options}
placeholder="selecione" />);

    const select = screen.getByPlaceholderText('selecione');
    expect(select).toBeInTheDocument();
    // alteramos o segundo argumento da seleção da opção
    // para pegar a partir do texto Usuário
    userEvent.selectOptions(select, screen.getByText('Usuário'));
    expect(select).toHaveValue('USER');
});
```

Dessa forma, garantimos que estamos selecionando o elemento a partir do seu texto, como de fato uma pessoa faria no componente.

Com isso, os testes do nosso input já estão bem bacanas!

A seguir, vamos testar um componente um pouco diferente, que possui algumas animações e é renderizado de uma forma um pouco particular.

Snackbar e suas animações

O componente de snackbar tem uma renderização um pouco diferente. Ele possui algumas animações de entrada/saída da tela que são controladas pela função `setTimeout`. Basicamente, seu funcionamento é o seguinte:

- O componente é renderizado na tela, mas não fica visível de início;
- 500 milissegundos depois, ele fica visível, com uma transição;
- 5 segundos depois (ou 5000 milissegundos) ele fica invisível novamente, mas ainda fica na tela;
- 500 milissegundos depois, ele é removido da tela.

Fora isso, existe um botão de X, que, ao ser clicado, também remove o snackbar da tela.

Esses são os cenários que precisamos testar: todas essas aparições/remoções da tela e o evento de clique. Vamos começar criando nosso arquivo `snackbar.unit.js` na pasta `src/__tests__/components` e importando os utilitários como na última vez e também importando o componente:

```
// arquivo src/__tests__/components/snackbar.unit.js
// importamos utilitário para consultas na tela
import { screen } from '@testing-library/react';
// importamos utilitário para disparo de eventos
import userEvent from '@testing-library/user-event';
// importamos utilitário para renderização do tema
import { renderWithTheme } from '../../testUtils';
// importamos o componente que será testado
import Snackbar from '../../components/snackbar';
```

Agora vamos criar nosso bloco `describe` / `it` para o nosso primeiro cenário, onde vamos testar o comportamento padrão sem o clique do botão que fecha o snackbar:

```
describe('<Snackbar />', () => {
  it('Renderiza e remove o snackbar com delay', () => {
  });
});
```

Vamos começar renderizando nosso snackbar com um texto qualquer e fazendo uma consulta pelo seu texto:

```
it('Renderiza e remove o snackbar com delay', async () => {
    // renderizamos snackbar com o texto "mensagem"
    renderWithTheme(<Snackbar>mensagem</Snackbar>);
    // fazemos uma consulta por esse texto
    const conteudo = screen.getByText('mensagem');
});
```

Podemos verificar que esse conteúdo está na tela, mas ainda não está visível, através das asserções `toBeInTheDocument` e `not.toBevisible` :

```
it('Renderiza e remove o snackbar com delay', async () => {
    renderWithTheme(<Snackbar>mensagem</Snackbar>);
    const conteudo = screen.getByText('mensagem');

    // assegura que está na tela
    expect(conteudo).toBeInTheDocument();
    // assegura que ainda não está visível
    expect(conteudo).not.toBeVisible();
});
```

Precisaremos trabalhar com os utilitários de temporizadores (ou `timers`) do Jest. Existem diversas funções que podemos utilizar. Para isso, precisamos ativar esses `timers` através da função `jest.useFakeTimers()` , que substituirá funções, como `setTimeout` , por mocks do Jest. Como precisaremos trabalhar em ambos os testes, podemos colocar isso em um bloco `beforeEach` , antes dos nossos testes:

```
describe('<Snackbar />', () => {
    // criamos um beforeEach
    beforeEach(() => {
        // inserimos a função jest.fakeTimers
        jest.useFakeTimers();
    });
// restante do código omitido
```

Por boas práticas e para manter nosso teste organizado, evitando que algum mock indesejado seja mantido caso tenhamos algum teste que utilize timers reais, podemos remover essa implementação de temporizadores mockados usando a função `jest.useRealTimers`. É interessante colocá-la no `afterEach` também:

```
describe('<Snackbar />', () => {
  beforeEach(() => {
    jest.useFakeTimers();
  });
  // criamos um afterEach
  afterEach(() => {
    // redefinimos os timers para os valores reais
    jest.useRealTimers();
  });
// restante do código omitido
```

Dentro do nosso teste, podemos agora executar a função `jest.advanceTimersByTime` para avançar uma determinada quantidade de tempo nos temporizadores. Como sabemos que o snackbar só fica visível após 500ms, vamos realizar um avanço com esse valor de 500:

```
it('Renderiza e remove o snackbar com delay', async () => {
    renderWithTheme(<Snackbar>mensagem</Snackbar>);
    const conteudo = screen.getByText('mensagem');

    expect(conteudo).toBeInTheDocument();
    expect(conteudo).not.toBeVisible();

    // avançamos 500ms
    jest.advanceTimersByTime(500);
});
```

Ao inserir essa linha, teremos um pequeno alerta no terminal, pois, como o React renderiza os componentes de forma assíncrona, alterar esse timer causou uma re-renderização no nosso componente, deixando-o visível na tela. Podemos utilizar a função `waitFor` da Testing Library para nos ajudar com esse cenário. Ela foi

feita para lidar com esses cenários de re-renderização de componente. Basta importá-la no começo do nosso arquivo:

```
// importamos waitFor
import { screen, waitFor } from '@testing-library/react';
```

E executá-la em nosso teste, englobando a função que avança o nosso temporizador:

```
it('Renderiza e remove o snackbar com delay', async () => {
    renderWithTheme(<Snackbar>mensagem</Snackbar>);
    const conteudo = screen.getByText('mensagem');

    expect(conteudo).toBeInTheDocument();
    expect(conteudo).not.toBeVisible();

    // englobamos com waitFor
    await waitFor(() => {
      jest.advanceTimersByTime(500);
    });
});
```

Vale ressaltar que essa função é assíncrona, por isso utilizamos `async` no callback de nosso teste e também utilizamos `await` antes dela.

Como vamos precisar fazer esse processo de esperar um temporizador com `waitFor`, podemos criar uma função para nos poupar algumas linhas de código. Vamos jogar todo esse bloco para uma função chamada `waitForTimersByTime`, que receberá o tempo como parâmetro e realizará tudo o que fizemos agora. Podemos criá-la fora do nosso bloco de testes:

```
const waitTimersByTime = async time => {
  await waitFor(() => {
    jest.advanceTimersByTime(time);
  });
};
```

E a seguir, vamos chamar essa função no teste:

```
  it('Renderiza e remove o snackbar com delay', async () => {
    renderWithTheme(<Snackbar>mensagem</Snackbar>);
    const conteudo = screen.getByText('mensagem');

    // verifica se está na tela, mas ainda não está visível
    expect(conteudo).toBeInTheDocument();
    expect(conteudo).not.toBeVisible();

    // modificamos para a função waitTimersByTime
    await waitTimersByTime(500);
});
```

Pode parecer algo desnecessário, mas como vamos executar essa ação pelo menos mais duas vezes, nosso teste fica um pouco mais limpo e legível. Nunca se esqueça de que teste também é código! Criar algumas abstrações pode ser algo muito útil também.

Agora que já avançamos nosso temporizador, podemos nos assegurar de que o elemento está visível em tela com a função `toBeVisible` (sem o `not`, que colocamos anteriormente):

```
it('Renderiza e remove o snackbar com delay', async () => {
    renderWithTheme(<Snackbar>mensagem</Snackbar>);
    const conteudo = screen.getByText('mensagem');

    expect(conteudo).toBeInTheDocument();
    expect(conteudo).not.toBeVisible();

    await waitTimersByTime(500);
    // asserção que verifica que o elemento está visível
    expect(conteudo).toBeVisible();
});
```

Precisaremos avançar nosso temporizador por mais cinco segundos (ou seja, 5000 milissegundos) e verificar que o elemento não está visível novamente:

```
it('Renderiza e remove o snackbar com delay', async () => {
    renderWithTheme(<Snackbar>mensagem</Snackbar>);
    const conteudo = screen.getByText('mensagem');
```

```
    expect(conteudo).toBeInTheDocument();
    expect(conteudo).not.toBeVisible();

    await waitTimersByTime(500);
    expect(conteudo).toBeVisible();

    // avançamos mais 5s/5000ms
    await waitTimersByTime(5000);
    // verificamos que não está mais visível
    expect(conteudo).not.toBeVisible();
});
```

Por último, precisamos avançar os últimos 500 milissegundos e verificar que o elemento foi removido. Podemos fazer isso com a função `not.toBeInTheDocument` :

```
it('Renderiza e remove o snackbar com delay', async () => {
    renderWithTheme(<Snackbar>mensagem</Snackbar>);
    const conteudo = screen.getByText('mensagem');

    expect(conteudo).toBeInTheDocument();
    expect(conteudo).not.toBeVisible();

    await waitTimersByTime(500);
    expect(conteudo).toBeVisible();

    await waitTimersByTime(5000);
    expect(conteudo).not.toBeVisible();

    // avançamos mais 500 milissegundos
    await waitTimersByTime(500);
    // verificamos que o elemento foi removido do DOM
    expect(conteudo).not.toBeInTheDocument();
});
```

Ufa, bastante coisa! Mas agora, nosso teste corresponde exatamente ao funcionamento do nosso componente. Existem várias funções e utilitários de temporizadores do Jest, mas, para o nosso caso, a função `advanceTimersByTime` serviu muito bem. É possível rodar todos os timers com a função `jest.runAllTimers()` , ou

rodar só os timers pendentes com a função `jest.runOnlyPendingTimers()` e algumas outras. Caso tenha interesse ou necessidade, você pode dar uma olhada na documentação oficial (https://jestjs.io/docs/pt-BR/timer-mocks). Você pode inserir a função `waitTimersByTime`, que fizemos no arquivo `testUtils`, caso queira manter a organização dos utilitários e se houver a necessidade de reutilizá-la.

Para terminarmos os testes do snackbar, falta somente verificarmos que, ao clicar no botão, o componente também sumirá da tela. Vamos criar um novo bloco com esse teste. Vamos colocar `async` na assinatura do callback do teste, pois precisaremos trabalhar com algumas re-renderizações do componente:

```
it('Remove o snackbar ao clicar no botão', async () => {
});
```

Vamos renderizar nosso snackbar e procurar pelo botão dentro dele:

```
it('Remove o snackbar ao clicar no botão', async () => {
    // renderizamos o snackbar
    renderWithTheme(<Snackbar>mensagem</Snackbar>);
    // consultamos nosso botão
    const botao = screen.queryByRole('button');
});
```

Podemos também procurar pela nossa mensagem e verificar que ela está na tela:

```
it('Remove o snackbar ao clicar no botão', async () => {
    renderWithTheme(<Snackbar>mensagem</Snackbar>);
    const botao = screen.getByRole('button');
    // consultamos algo pela mensagem
    const mensagem = screen.queryByText('mensagem');
    // asseguramos que está na tela
    expect(mensagem).toBeInTheDocument();
});
```

O que precisamos fazer a seguir é disparar nosso evento de click nesse botão:

```
it('Remove o snackbar ao clicar no botão', async () => {
    renderWithTheme(<Snackbar>mensagem</Snackbar>);
    const botao = screen.getByRole('button');

    userEvent.click(botao);
});
```

E verificar que o elemento não está mais na tela. Vamos aproveitar para aprender uma outra função da Testing Library, a `waitForElementToBeRemoved`. Podemos utilizá-la em casos como esse, em vez de `not.toBeInTheDocument`. Vamos importá-la no início do nosso arquivo:

```
import { screen, waitFor, waitForElementToBeRemoved } from '@testing-library/react';
```

Para utilizá-la em nosso teste, basta executá-la passando como argumento uma função com a consulta do elemento que esperamos que seja removido. Podemos passar a consulta da nossa própria mensagem ou botão, por exemplo:

```
it('Remove o snackbar ao clicar no botão', async () => {
    renderWithTheme(<Snackbar>mensagem</Snackbar>);
    const botao = screen.getByRole('button');
    const mensagem = screen.queryByText('mensagem');

    expect(mensagem).toBeInTheDocument();

    userEvent.click(botao);
    // executamos a função waitForElementToBeRemoved
    // consultando pelo elemento da mensagem
    await waitForElementToBeRemoved(() => screen.queryByText('mensagem'));
});
```

Com isso, nosso teste também já garante que a funcionalidade de fechar o snackbar ocorre corretamente!

Carrossel e sua lista de perfis

Vamos testar o componente de carrossel, responsável por renderizar uma lista de cards de perfil do componente Profile. Nesses cenários, existem duas formas de testar o componente:

- Realizando um mock do componente de Profile.
- Não realizando o mock.

À primeira vista, pode ser natural tentar testar os comportamentos do componente de Profile já por aqui (em uma pequena espécie de teste de integração), mas lembre-se de que, nesse momento, o componente que está sendo testado é o componente `Carousel`.

É justamente por isso que vamos realizar o mock do Profile nesse teste, faz bastante sentido seguirmos essa abordagem e realizar um mock desse componente já que haverá futuramente um teste específico só para ele. Fazendo isso, garantimos que o componente de carrossel apenas renderize uma lista desse determinado componente sem nos preocuparmos com o funcionamento dos perfis por si só, isolando nossos testes e fazendo com que cada um cuide de uma unidade em si.

Dito isso, vamos criar o arquivo `carousel.unit.js` dentro da pasta `src/__tests__/components` e importar o que precisamos para começar nosso teste:

```
// arquivo src/__tests__/componentes/carousel.unit.js
// utilitário que nos permite fazer consulta na tela
import { screen } from '@testing-library/react';
// utilitário de renderização com tema
import { renderWithTheme } from '../../testUtils';
// componente que será testado
import Carousel from '../../components/carousel';
```

Agora, vamos realizar o mock do componente de Profile, como já vimos anteriormente nos outros testes que fizemos. Vamos fazer com que ele retorne apenas uma `div` e, para nos auxiliar nos testes, vamos colocar um atributo `data-testid` para consultarmos esse elemento futuramente:

```
import { screen } from '@testing-library/react';
import { renderWithTheme } from '../../testUtils';

import Carousel from '../../components/carousel';
// Mockamos o componente de Profile
//indicando a div com data-testid como retorno
jest.mock('../../components/profile', () => () => <div data-
testid="profile" />);
```

Se você notou, tivemos que mockar uma função com `jest.mock` e, dentro do retorno, retornar outra função, por isso a dupla sequência de arrow functions `() => () =>` ali.

Isso acontece pelo fato de exportarmos o componente como `default` (em vez de exportar pelo seu nome, com `named exports`) usando o padrão de ECMAScript Modules dentro de um projeto CommonJS (caso queira rever os diferentes tipos de módulos na linguagem JavaScript, você pode ler esse post https://gabrieluizramos.com.br/modulos-em-javascript).

A outra alternativa seria escrever um mock assim:

```
jest.mock('../../components/profile', () => ({
  __esModule: true,
  default: () => <div data-testid="profile" />
}));
```

No qual indicamos que o módulo usa módulos ECMAScript (`__esModule`) e indicamos o mock para o valor `default`. Mas fica muito mais verboso, não acha?

Isso tudo é necessário porque esse projeto, diferente dos outros, não está com a configuração de `"type": "module"` habilitada em seu `package.json` e, por padrão, qualquer projeto sem isso é executado como CommonJS pelo Node.

Por se tratar de um projeto que possui um sistema de compilação/build dos componentes criado com o `create-react-app` (que utiliza Webpack/Babel por baixo dos panos), toda essa "magia"

de configuração acontece de forma transparente para quem está desenvolvendo. É um detalhe que não impacta muito nosso fluxo de trabalho, mas é importante comentar. Vamos voltar ao que importa!

Agora podemos iniciar nosso teste. Vamos criar nosso bloco `describe` / `it` :

```
describe('<Carousel />', () => {
    it('Deve renderizar uma lista de perfis', () => {
    });
});
```

O carrossel recebe algumas propriedades como:

- `items` : uma lista (array de objetos) de elementos para renderizar como perfil;
- `editable` : um booleano que indica se os perfis podem ser editados ou não;
- `onClickDelete` : uma função que é disparada ao clicar no botão deletar do perfil;
- `onClickEdit` : uma função que é disparada ao clicar no botão editar do perfil.

Desses quatro valores, três deles (`editable` , `onClickDelete` e `onClickEdit`) são apenas repassados ao componente de perfil e não interessam muito aos nossos testes, já que estamos testando o carrossel em si e realizando o mock do componente de perfil. Então vamos criar uma variável fora do nosso bloco `describe` com o nome de `BASE_PROPS` , onde deixaremos esses valores, apenas para reutilizarmos:

```
// criamos uma variável com alguns valores
// apenas para reutilizarmos
const BASE_PROPS = {
  editable: true,
  onClickEdit: jest.fn(),
  onClickDelete: jest.fn()
};
```

```
describe('<Carousel />', () => {
  it('Deve renderizar uma lista de perfis', () => {
  });
});
```

Dentro do nosso teste, precisamos criar uma estrutura para simular a propriedade `items`. Como o carrossel repassa todos os valores de cada item dessa lista para um componente de perfil, a única coisa de que precisamos é um objeto com o atributo `name`, que é utilizado ao realizar o map dentro do componente de carrossel.

Vamos criar essa lista dentro do nosso primeiro teste. Vamos criar, por exemplo, uma lista com três itens:

```
it('Deve renderizar uma lista de perfis', () => {
    // criamos lista de itens
    const items = [
      { name: 'primeiro' },
      { name: 'segundo' },
      { name: 'terceiro' },
    ];
});
```

E vamos renderizar o componente de carrossel, passando esses itens e as props que criamos fora do nosso teste (`BASE_PROPS`):

```
  it('Deve renderizar uma lista de perfis', () => {
    const items = [
      { name: 'primeiro' },
      { name: 'segundo' },
      { name: 'terceiro' },
    ];
    // renderizamos o Carrossel com todos os items
    // e fazendo spread da variável BASE_PROPS
    // passando todos os seus valores
    renderWithTheme(<Carousel items={items} {...BASE_PROPS} />);
  });
```

Podemos realizar duas consultas para garantir que nosso componente está renderizando tudo o que deveria. Como o carrossel renderiza uma lista com vários itens, é interessante

consultar se essa lista existe (pela role `list`), se existem três itens dessa lista (com a role `listitem`) conforme nosso mock de `items` e, também, se existem três tags `div` com `test id` de `profile`, como nosso mock. Vamos lá:

```
it('Deve renderizar uma lista de perfis', () => {
    const items = [
      { name: 'primeiro' },
      { name: 'segundo' },
      { name: 'terceiro' },
    ];
    renderWithTheme(<Carousel items={items} {...BASE_PROPS} />);
    // consultamos a lista
    const list = screen.getByRole('list');
    // consultamos os perfis por testId
    const profiles = screen.getAllByTestId('profile');
    // consultamos os itens por sua role
    const itens = screen.getAllByRole('listitem');
});
```

Note que, agora, para consultar vários elementos por um mesmo seletor, utilizamos as consultas com `getAll`.

Já podemos verificar se a lista está no documento corretamente. Podemos validar também a existência de três perfis e itens (já que foram as `props` que fornecemos):

```
it('Deve renderizar uma lista de perfis', () => {
    const items = [
      { name: 'primeiro' },
      { name: 'segundo' },
      { name: 'terceiro' },
    ];
    renderWithTheme(<Carousel items={items} {...BASE_PROPS} />)
    const lista = screen.getByRole('list');
    const perfis = screen.getAllByTestId('profile');
    const itens = screen.getAllByRole('listitem');

    // verificamos que a lista está na tela
    expect(lista).toBeInTheDocument();
```

```
    // verificamos que existem 3 perfis
    expect(perfis.length).toEqual(3);
    // verificamos que existem 3 itens também
    expect(itens.length).toEqual(3);
});
```

Assim nosso teste garante que uma lista seja renderizada corretamente. Seria interessante também garantir que os botões que ajudam o usuário na navegação da lista sejam exibidos. Vamos consultar esses elementos e inserir uma asserção para eles:

```
it('Deve renderizar uma lista de perfis', () => {
    const items = [
      { name: 'primeiro' },
      { name: 'segundo' },
      { name: 'terceiro' },
    ];
    renderWithTheme(<Carousel items={items} {...BASE_PROPS} />)
    const lista = screen.getByRole('list');
    const perfis = screen.getAllByTestId('profile');
    const itens = screen.getAllByRole('listitem');
    // consultamos todos os botões
    const botoes = screen.getAllByRole('button');

    expect(lista).toBeInTheDocument();
    expect(perfis.length).toEqual(3);
    expect(itens.length).toEqual(3);
    // asseguramos que existem somente dois
    expect(botoes.length).toEqual(2);
});
```

Bem prático, não? Com isso, é provável que uma dúvida relacionada ao `data-testid` tenha surgido e vale a pena respondê-la.

Devo utilizar test-id em todos os meus componentes?

Na minha opinião (e até na opinião do Kent C. Dodds, criador da biblioteca Testing Library), assim como respondi sobre o snapshot, a resposta é "não".

Se você reparou em como estamos conduzindo nossos testes, eles seguem um fluxo bem simples:

- Renderizamos nosso componente;
- Asseguramos que seus elementos necessários estão em tela;
- Disparamos alguma ação como `click` ou algo do tipo, quando necessário;
- Verificamos se essa ação ocorreu e o resultado que ela produziu em tela.

Estamos, de forma unitária, simulando uma pessoa interagindo com o nosso componente. Um usuário não vai acessar um elemento através do seu atributo `data-testid` , ele vai interagir com algo visível na tela. Nesse caso, como realizamos um mock e o componente de Profile terá seus próprios testes garantindo sua qualidade, não há problema já que, do ponto de vista do componente de carrossel, ele apenas exibe uma lista de outros elementos.

Portanto, nada de sair colocando `data-testid` em tudo por aí só para facilitar os testes, ok?

"Quanto mais seus testes se assemelham à maneira como seu software é utilizado, mais confiança eles podem lhe dar." - Kent C. Dodds

Acho interessante tirarmos um momento para entender essa frase e ver como esse princípio está ligado ao que estamos fazendo até o momento.

Até aqui, tudo o que fizemos foi testar o nosso sistema da maneira que alguém interagiria com ele. Seja no front-end ou no back-end, aplicamos esse princípio mesmo sem termos nos dado conta disso. Interessante, não?

Esse é um dos princípios que guiam inclusive a biblioteca de testes que estamos utilizando (https://testing-library.com/docs/guiding-principles/). Essa frase é bem conhecida na comunidade quando o assunto é testes e é frequentemente dita por Kent C. Dodds em

seus conteúdos (https://twitter.com/kentcdodds/status/977018512689455106).

Acredito que essa é uma pequena reflexão muito bem-vinda nesse momento da nossa jornada para que esse tipo de pensamento possa guiar nossos testes e nos ajude a alcançar um nível bacana de qualidade de código.

Com tudo isso, testamos metade dos componentes da aplicação e aprendemos bastante coisa: a renderizar elementos, a disparar eventos, a consultar elementos em tela e até a fazer mock de outros subcomponentes usados dentro de um componente específico! A cada passo, estamos garantindo qualidade de um pedaço das nossas aplicações.

Deixo como desafio desta parte realizar os testes dos quatro componentes restantes da aplicação:

- `Avatar` : componente que exibe uma imagem de um usuário (ou um ícone, caso não tenha imagem cadastrada), podendo ou não ter uma variante minimalista através da propriedade `minimal` . Pode ser um caso onde o `toMatchSnapshot` auxilie bastante;
- `Card` : uma div com uma borda/sombra que serve para montar os perfis e os formulários da aplicação;
- `Profile` : componente utilizado dentro do carrossel, que exibe os dados de um usuário que pode ou não ser editável (através da propriedade `editable`) e possuir ações de click (`onClickEdit` e `onClickDelete`). Precisaremos simular essas ações também;
- `Form` : um componente que recebe uma propriedade chamada `schema` , que possui a configuração de campos de um formulário. Ele também recebe uma propriedade `onSubmit` , que só é disparada se todos os campos do formulário que possuem validações estiverem corretamente preenchidos. Precisaremos validar esse comportamento.

Dentro da pasta `src/mocks`, existem alguns dados que podem ajudar você a simular dados de perfis para os componentes acima e para o formulário também.

Com tudo o que vimos até aqui sei que você conseguirá testar esses componentes sem problemas.

Aproveito também para deixar este post do próprio Kent C. Dodds caso você tenha curiosidade de ler, sobre alguns erros comumente cometidos por quem utiliza essa biblioteca (https://kentcdodds.com/blog/common-mistakes-with-react-testing-library) e também sua versão traduzida por Willian Justen (https://willianjusten.com.br/erros-comuns-com-o-react-testing-library/).

Agora, vamos testar alguns utilitários da nossa aplicação front-end.

11.3 Testando utilitários de clientes

Testar o restante da aplicação antes de partirmos para os testes de integração será bem tranquilo e você vai ver que é bem parecido com os testes unitários que fizemos, principalmente quando testamos a aplicação em Node/Express.

Vamos começar testando um dos clientes de HTTP que são utilizados para fazer requisições para a API. Vamos criar a pasta `clients` dentro de `src/__tests__/` e, dentro dela, a pasta `http`. Começaremos pelo arquivo responsável pela autenticação, então vamos criar o arquivo `authentication.unit.js` e importar tudo desse cliente para que possamos testar também:

```
// arquivo src/__tests__/clients/http/authentication.unit.js
// importamos todas as funções do cliente como "auth"
import * as auth from '../../../clients/http/authentication';
```

Esse cliente exporta as funções relacionadas à autenticação, como a de `logIn`, que realiza um `POST` no endpoint de autenticação da API e depois chama a função `setUserData` do cliente de `storage`, para salvar algumas informações localmente no navegador. Como vamos precisar validar se o cliente de `storage` foi chamado, vamos realizar seu mock e importá-lo também:

```
import * as auth from '../../../clients/http/authentication';

// importamos o cliente de storage
import * as storage from '../../../clients/storage';
// mockamos as funções de storage
jest.mock('../../../clients/storage.js');
```

Vamos criar um teste para a função de `login` começando pelos blocos `describe` / `it`:

```
describe('Cliente de autenticação', () => {
  it('Realiza login e armazena os dados localmente', async () => {
  });
});
```

Precisamos executar a função de login e garantir que ela chame o endpoint correto de autenticação. Para facilitar, os endpoints da aplicação podem ser acessados no arquivo `clients/http/endpoints.js`, que basicamente exporta os endpoints com constantes para que não precisemos ficar manipulando as strings diretamente no código e nos nossos testes. Vamos importá-lo:

```
import * as auth from '../../../clients/http/authentication';

import * as storage from '../../../clients/storage';
jest.mock('../../../clients/storage.js');

// importamos os endpoints
import endpoints from '../../../clients/http/endpoints';

describe('Cliente de autenticação', () => {
```

```
  it('Realiza login e armazena os dados localmente', async () => {});
});
```

Agora podemos executar a função `auth.logIn`. Essa função aplica uma técnica interessante para auxiliar nos nossos testes. Vamos dar uma olhada em sua assinatura:

```
// arquivo src/clients/http/authentication.js
import baseClient from './config';
import * as storage from '../storage';
import endpoints from './endpoints';

export const logIn = async ({
  client = baseClient,
  username,
  password
}) => {
  const user = await client.post(endpoints.authentication, { username,
password });
  storage.setData(user);

  return user;
};
```

Em sua assinatura, percebemos que ela recebe um objeto que pode receber as chaves `client`, `username` e `password`. Note que, caso `client` não seja fornecido, seu valor padrão será o valor de `baseClient`, que é o cliente do axios utilizado na aplicação.

O nome dessa "estratégia", de receber dependências como argumento, é **injeção de dependência** e pode nos auxiliar a escrever um código mais testável em cenários como esse. O que acontece nesse caso é que nossa função **depende** de um cliente HTTP para fazer uma requisição (no caso, o axios). Existe um cliente padrão (**baseClient**) que é utilizado caso nenhum valor de **client** seja passado, porém, nos nossos testes, podemos fornecer um valor customizado para facilitar nossas asserções.

Se quiser ler um pouco mais sobre injeção de dependência, eu fiz um post bem breve sobre o assunto há algum tempo

(https://gabrieluizramos.com.br/desmistificando-injecao-de-dependencia/), e existem várias referências interessantes por aí na internet também.

Sendo assim, o que precisamos fazer para testar nossa função é basicamente criar um mock desse cliente e passá-lo junto aos dados de `username` e `password`. Vamos lá, começando pelo mock do cliente. Podemos usar os dados de `src/mocks/profile` para nos auxiliar a fazer o mock de seu retorno:

```
import * as auth from '../../../clients/http/authentication';

import * as storage from '../../../clients/storage';
jest.mock('../../../clients/storage.js');

import endpoints from '../../../clients/http/endpoints';

// importamos os dados de perfil e renomeamos para `mockProfile`
import { profile as mockProfile } from '../../mocks/profile';

describe('Cliente de autenticação', () => {
  it('Realiza login e armazena os dados localmente', async () => {
    // criamos cliente falso
    const client = {
      // com função get que resolverá com os dados de mock
      post: jest.fn().mockResolvedValue(mockProfile)
    };
  });
});
```

Com isso, podemos passar a executar a função `auth.logIn` fornecendo esse cliente e alguns dados. Conseguimos reusar o `userName` do mock que importamos, mas não temos dados de senha por lá, então podemos passar manualmente. Vamos salvar seu retorno em uma variável também:

```
it('Realiza login e armazena os dados localmente', async () => {
    const client = {
      post: jest.fn().mockResolvedValue(mockProfile)
    };
```

```
    // criamos senha qualquer
    const password = 'senha super secreta';
    // executamos a função de login fornecendo os dados de usuário
    // e armazenamos seu retorno na variável user
    const user = await auth.logIn({
      // username do mock
      username: mockProfile.userName,
      // senha criada
      password,
      // cliente que criamos
      client
    });
});
```

Em seguida, podemos verificar se a função `client.post` foi chamada corretamente uma única vez e com os dados esperados. Podemos também verificar se os dados retornados são os dados de `mockProfile` que esperamos:

```
it('Realiza login e armazena os dados localmente', async () => {
    const client = {
      post: jest.fn().mockResolvedValue(mockProfile)
    };

    const password = 'senha super secreta';

    const user = await auth.logIn({
      client,
      username: mockProfile.userName,
      password
    });

    // verificamos se os dados retornados
    // são iguais ao mockprofile
    expect(user).toEqual(mockProfile);
    // verificamos se client.post foi chamada
    // uma única vez
    expect(client.post).toHaveBeenCalledTimes(1);
    // e com o endpoint de autenticação
    // e os dados de usuário
```

```
    expect(client.post).toHaveBeenCalledWith(endpoints.authentication, {
      username: mockProfile.userName,
      password
    });
});
```

Para finalizar, também é interessante garantir que a função `storage.setData` tenha sido chamada uma única vez e com os dados do usuário também, então vamos criar essas asserções:

```
it('Realiza login e armazena os dados localmente', async () => {
    const client = {
      post: jest.fn().mockResolvedValue(mockProfile)
    };

    const password = 'senha super secreta';

    const user = await auth.logIn({
      client,
      username: mockProfile.userName,
      password
    });

    expect(user).toEqual(mockProfile);

    expect(client.post).toHaveBeenCalledTimes(1);
    expect(client.post).toHaveBeenCalledWith(endpoints.authentication, {
      username: mockProfile.userName,
      password
    });

    // asseguramos que a função storage.setData
    // foi chamada uma única vez
    expect(storage.setData).toHaveBeenCalledTimes(1);
    // e com os dados retornados do usuário
    expect(storage.setData).toHaveBeenCalledWith(user);
});
```

Pronto, nosso teste já simula o comportamento da função que autentica um usuário!

Bem mais simples utilizar injeção de dependência para esses cenários. Conseguiríamos testar sem isso, mas teríamos que realizar um mock do `baseClient`, dos seus métodos e do seu retorno. Da forma como fizemos, nosso teste fica um pouco mais limpo e organizado, e também evita esse trabalho extra de configuração. Isso fica ainda mais evidente quando as dependências são valores globais, como `window` ou `document`.

As funções restantes desse arquivo são apenas reexportações das funções de storage, então não precisamos nos preocupar em testá-las agora. Deixo como desafio desta parte para você fazer os testes do cliente que realiza as requisições relacionadas aos perfis de usuários, dentro da pasta `clients/http/profiles`. A lógica será exatamente a mesma que fizemos até aqui. Após isso, o arquivo `clients/http/storage` possui algumas funções bem simples de serem testadas e que também seguem a mesma lógica de injeção de dependência para os clientes de storage, que são objetos um pouco mais trabalhosos de se ficar fazendo mock e atualizando.

Com isso, vamos testar mais um pedaço da nossa aplicação, as funções responsáveis por gerenciar todo o estado global.

11.4 Testando estado global

Como comentamos no início deste capítulo, essa aplicação foi desenvolvida com Redux para gerenciamento de estado global. Ok... mas, caso você não tenha experiência com React ou Redux, isso pode soar um pouco estranho para você. Afinal, o que é estado? E o que é um estado global?

Podemos pensar em estado como os valores internos dos nossos componentes que são representados em uma interface. Vamos pensar em um exemplo bem simples: um botão com um contador. Toda vez que um clique ocorre, esse botão soma + 1 em seu

contador. Esse valor de contador também é exibido na tela. É um exemplo bem simples, mas que pode nos auxiliar nessa tarefa de entender o que é estado. Nesse caso, o valor interno do componente que é representado em tela é esse próprio valor de contagem que é incrementado a cada clique.

Como **estado** está muito ligado aos dados representados em tela e a sua visualização como componente de interface, muitas vezes escrever componentes que possuem estado de forma reutilizável pode ser uma tarefa ligeiramente mais complexa. A ideia de um gerenciador de estados global é, justamente, manter toda essa camada de dados desatrelada do componente em si. Dessa forma, é possível criar uma abstração onde os dados ficam armazenados (`store`) e, através de ações devidamente mapeadas (`actions`), novos valores de estados são gerados (através de `reducers`). Os componentes que necessitam desses valores simplesmente conectam-se a essa abstração, que armazena tudo globalmente, e, dada alguma mudança, realizam suas determinadas re-renderizações.

Parece complexo, mas podemos resumir todo esse fluxo pensando na seguinte divisão:

- Uma aplicação com Redux possui seus componentes mais básicos de interface, como os componentes que testamos até aqui;
- Todo estado principal da aplicação não fica em um componente em específico, mas sim, fica armazenado em um objeto que pode ser acessado globalmente, chamado `store` ;
- Os componentes que precisarem podem acessar os dados da `store` quando necessário;
- Caso alguma modificação nesse estado precise ocorrer (carregar alguma informação ou mudar algo após um evento), ela deve ser feita através de `actions` mapeadas que podem ser disparadas pela função `dispatch` ;
- Para gerar um novo estado, existem `reducers` , funções que recebem um estado prévio e uma ação disparada e são

responsáveis por gerar um novo estado.

Existem algumas nomenclaturas já utilizadas na comunidade para distinguir os componentes, já que qualquer um pode, em teoria, conectar-se aos dados "globais". É comum dividi-los em algumas categorias, sendo os que se conectam com esses dados (fornecendo uma certa camada de lógica e acesso à esses valores) ou não (servindo mais "puramente" apenas para renderização da interface). Alguns nomes que você pode encontrar por aí são:

- Componentes "contêineres" (*containers*) e "apresentacionais" (*presentational*).
- Componentes "inteligentes" (*smart*) e "burros" (*dumb*).

Contêineres/inteligentes são componentes que se conectam a esse estado global, e apresentacionais/burros apenas renderizam algo simples. Isso serve para que possamos isolar as responsabilidades de nossos componentes de forma um pouco mais clara.

Na nossa aplicação, os componentes contêineres são as páginas (dentro de `src/pages`) que acessam os dados da *store* do Redux, criam as funções que disparam as ações corretamente e as passam como propriedade para os componentes (dentro de `src/components`) necessários.

Como tudo isso que vimos, basicamente, cria algumas camadas extras de abstração na nossa aplicação, podemos pensar que nossa aplicação está com mais ou menos a seguinte estrutura:

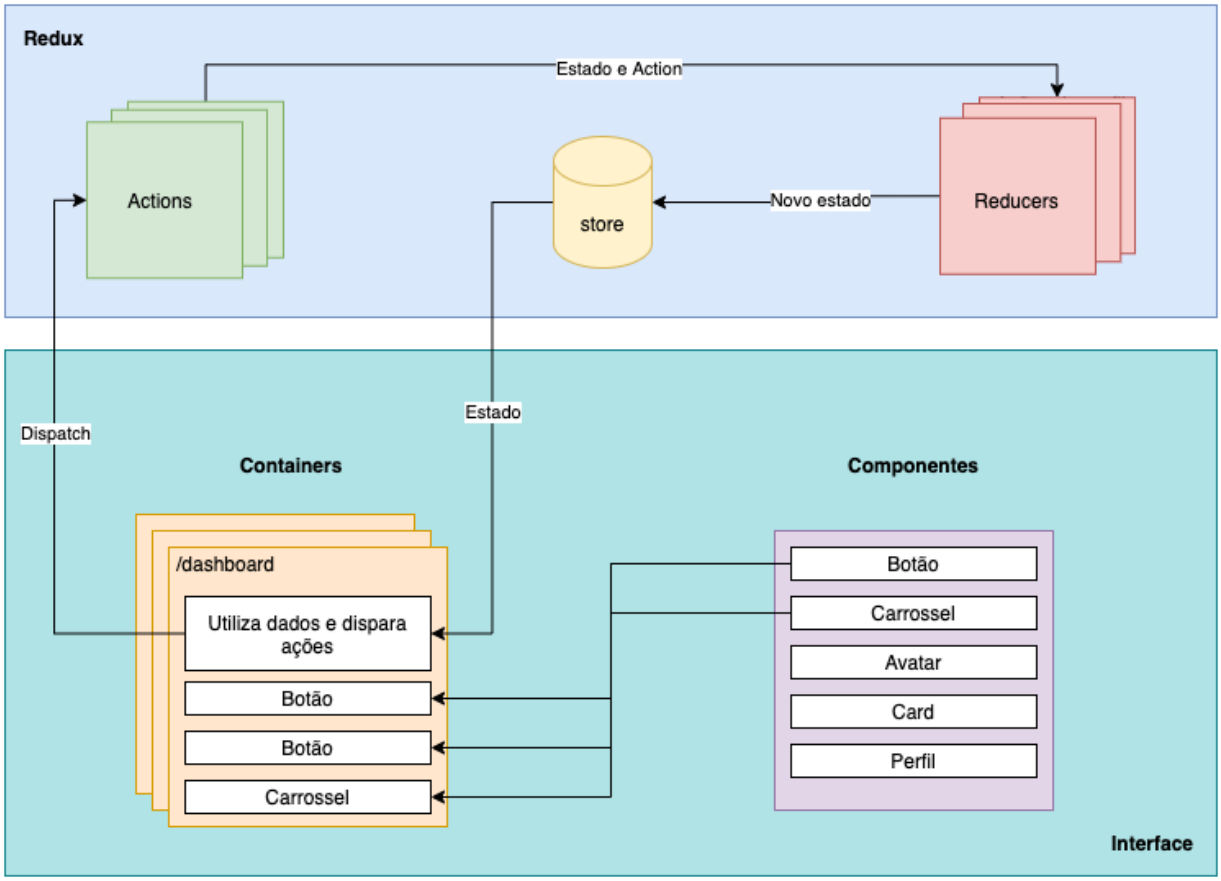


Figura 11.1: Aplicação front-end com Redux.

Somente alguns componentes (no caso, as páginas de login e de dashboard) conectam-se ao Redux, formatam algumas ações e montam os fluxos com os componentes mais básicos da pasta `components`.

No caso da nossa aplicação, alguns dados que ficam no estado global são:

- Dados sobre o usuário logado, como seu nível de permissão (`ADMIN` ou `USER`) e se está ou não autenticado;
- Dados sobre os perfis de usuários cadastrados;
- Dados sobre notificações, se deve ou não aparecer um snackbar com alguma mensagem.

E as ações existentes são justamente as ações que manipulam todos esses dados, realizando login/logout, atualizando e

removendo usuários e aplicando notificações na tela.

Depois de tudo isso, muito provavelmente você deve estar se perguntando: para que tudo isso?

Aplicar padrões como Redux muitas vezes pode parecer algo simplesmente complexo demais para uma aplicação e, sem dúvidas, nosso projeto se encaixa nesse cenário. Entretanto, como nosso foco aqui são os testes e estamos tentando aprender a maior gama de possibilidades, é interessante abordarmos Redux pelo fato de ser uma ferramenta muito utilizada no mercado, principalmente em aplicações React.

Os testes que realizaremos a seguir são somente dessa camada do Redux: testaremos suas ações, reducers e verificaremos se os estados são produzidos como o esperado. O fluxo completo de funcionamento das páginas deixaremos para os testes de integração, ok?

Testando actions e action creators

Vamos começar pelas ações de nossa aplicação. *Action creators* são apenas funções que nos auxiliam na tarefa de criar ações. Isso pode ser interessante, pois *actions*, em um padrão Redux, são apenas objetos com uma chave `type` descrevendo o que está ocorrendo na aplicação:

```
{
  type: 'AUTENTICANDO_USUARIO'
}
```

Então, podemos criar alguns testes para validar que essas ações são geradas devidamente.

Vamos realizar os testes da camada que cuida dos dados de usuário. Dentro de nossa pasta `src/__tests__` , vamos criar as pastas `store/user` e começar criando o arquivo `actions.unit.js` lá. Vamos

importar tudo o que esse arquivo exporta para nos poupar algumas linhas de código:

```
// arquivo src/__tests__/store/user/action.unit.js
import * as actions from '../../../store/user/actions';
```

Existem quatro funções que simplesmente criam novas actions: `authenticating`, para iniciar a autenticação de um usuário; `authenticateSuccess`, para armazenar as informações de uma autenticação bem-sucedida; `authenticateError`, para indicar uma autenticação com erro; e `clearAuthentication`, para limpar os dados de uma autenticação.

Começaremos testando essas quatro funções. Como são funções que basicamente possuem um objeto como saída, vamos criar nosso bloco `describe` com um `it.each`:

```
describe('Creators', () => {
  it.each([
    ['authenticating'],
    ['authenticateSuccess'],
    ['authenticateError'],
    ['clearAuthentication']
  ])('%s deve criar uma ação com o tipo %s', (creator) => {
  });
});
```

Agora, podemos elaborar cada caso de teste. Como a ideia é comparar se o objeto produzido possui um `type` igual a uma ação existente, vamos passar essas ações como os segundos argumentos no array dos cenários de testes. A ação `authenticating` tem que gerar um objeto com o tipo `AUTHENTICATE_PENDING`, a ação `authenticateSuccess` deve gerar um objeto com tipo `AUTHENTICATE_SUCCESS` e assim por diante:

```
describe('Creators', () => {
  it.each([
    // passamos as ações como segundo argumento
    // nos arrays
    ['authenticating', actions.AUTHENTICATE_PENDING],
```

```
    ['authenticateSuccess', actions.AUTHENTICATE_SUCCESS],
    ['authenticateError', actions.AUTHENTICATE_ERROR],
    ['clearAuthentication', actions.LOGOUT]
    // recebemos elas na função de callback
  ])('%s deve criar uma ação com o tipo %s', (creator, type) => {
  });
});
```

Fora isso, algumas dessas funções podem receber um argumento extra chamado `payload` com algum dado (por exemplo, as informações do usuário autenticado). Vamos passar esse terceiro argumento e, para os casos em que não temos nenhum `payload`, vamos informar o valor `undefined` explicitamente. Podemos importar o perfil de um usuário da pasta `mock` para nos ajudar com os dados:

```
// importamos o perfil de mock
import { profile } from '../../../mocks/profile';

describe('Creators', () => {
  it.each([
    // informamos os payloads como terceiro argumento
    ['authenticating', actions.AUTHENTICATE_PENDING, undefined],
    ['authenticateSuccess', actions.AUTHENTICATE_SUCCESS, profile],
    ['authenticateError', actions.AUTHENTICATE_ERROR, 'Ocorreu um erro'],
    ['clearAuthentication', actions.LOGOUT, undefined]
    // recebemos o payload na função de callback
  ])('%s deve criar uma ação com o tipo %s', (creator, type, payload) => {
  });
});
```

Agora podemos realizar nosso teste de fato. Como já configuramos tudo o que é necessário para executar uma ação pelos argumentos nos arrays, podemos simplesmente executar a action a partir da chave `creator` fornecendo o `payload` como argumento:

```
describe('Creators', () => {
  it.each([
    ['authenticating', actions.AUTHENTICATE_PENDING, undefined],
    ['authenticateSuccess', actions.AUTHENTICATE_SUCCESS, profile],
    ['authenticateError', actions.AUTHENTICATE_ERROR, 'Ocorreu um erro'],
```

```
    ['clearAuthentication', actions.LOGOUT, undefined]
  ])('%s deve criar uma ação com o tipo %s', (creator, type, payload) => {
    // executa a função a partir do nome fornecido
    // e informando o payload como argumento
    const action = actions[creator](payload);
  });
});
```

Em seguida, podemos criar nossa asserção validando se o campo `type` retornado é igual ao valor de `type`, assim como o `payload`:

```
describe('Creators', () => {
  it.each([
    ['authenticating', actions.AUTHENTICATE_PENDING, undefined],
    ['authenticateSuccess', actions.AUTHENTICATE_SUCCESS, profile],
    ['authenticateError', actions.AUTHENTICATE_ERROR, 'Ocorreu um erro'],
    ['clearAuthentication', actions.LOGOUT, undefined]
  ])('%s deve criar uma ação com o tipo %s', (creator, type, payload) => {
    const action = actions[creator](payload);
    // verificamos que o valor de action.type
    // é igual à variável type
    expect(action.type).toEqual(type);
    // verificamos que o valor de action.payload
    // é igual à variável payload
    expect(action.payload).toEqual(payload);
  });
});
```

Com isso, já temos os testes de nossas funções que criam as ações!

Testando Thunks

Algumas ações são um pouco mais complexas e podem envolver o disparo de várias outras pequenas ações. Por exemplo, realizar o login de um usuário pode envolver:

- Disparar a ação de `authenticating`;
- Tentar logar o usuário;

- Disparar a ação `authenticateSuccess` caso os dados sejam válidos, fornecendo os dados do usuário que a API retornou;
- Disparar a ação `authenticateError` caso os dados sejam inválidos, fornecendo a mensagem de erro.

Como essas ações são mais complexas e envolvem requisições HTTP, é comum deixá-las agrupadas em uma única ação que realizará todo o processo de forma assíncrona, comumente chamada de `thunk`, um middleware responsável por fazer essas ações assíncronas funcionarem corretamente.

Dentro do arquivo `src/store/user/actions.js`, existe um bloco com o comentário de título `thunks`. Dando uma olhada nas funções que estão lá, você consegue rapidamente perceber que, nesse projeto, uma função de `thunk` funciona da seguinte maneira:

- Ela recebe algum parâmetro com os dados que deve trabalhar (como `username` e `password`);
- Retorna uma outra função `async`, que recebe como parâmetro `dispatch`, uma função utilizada para executar os `action creators` que testamos;
- Realiza as chamadas HTTP dos clientes envolvidos, tratando seu retorno e disparando as devidas ações.

Com isso, vamos começar testando a função `thunk` de `login`. Vamos criar seu `describe` / `it` no mesmo arquivo em que já estávamos trabalhando:

```
describe('Login thunk', () => {
  it('Deve disparar as ações de login com sucesso', async () => {
  });
});
```

Como precisaremos realizar um mock do cliente HTTP que cuida da autenticação, vamos realizá-lo no início do nosso arquivo:

```
import * as auth from '../../../clients/http/authentication';
jest.mock('../../../clients/http/authentication');
```

Já vamos inserir em nosso `describe` o hook que limpará as execuções dessa função:

```
describe('Login thunk', () => {
  // inserimos o hooks
  afterEach(() => {
    jest.clearAllMocks()
  });

  it('Deve disparar as ações de login com sucesso', async () => {
  });
});
```

Agora vamos criar nosso teste para o cenário de sucesso. Como já possuímos os dados de um perfil de mock, vamos utilizá-lo como retorno da função `auth.logIn` , que realiza a chamada à API:

```
describe('Login thunk', () => {
  afterEach(() => {
    jest.clearAllMocks()
  });

  it('Deve disparar as ações de login com sucesso', async () => {
    // inserimos o mock resolvido com os dados de perfil
    auth.logIn.mockResolvedValueOnce(profile);
  });
});
```

E vamos criar uma função `dispatch` como um valor padrão de `jest.fn` para que possamos realizar nossas asserções:

```
it('Deve disparar as ações de login com sucesso', async () => {
    auth.logIn.mockResolvedValueOnce(profile);
    // criamos o dispatch
    const dispatch = jest.fn();
});
```

Tudo o que precisamos fazer é disparar a função de `thunk` corretamente, verificando se o cliente e o dispatch foram chamados como esperávamos. Podemos passar como `username` o nome de usuário de nosso mock e, como ele não possui uma senha no objeto

e já realizamos o mock do retorno esperado, podemos passar qualquer valor como senha:

```
it('Deve disparar as ações de login com sucesso', async () => {
    auth.logIn.mockResolvedValueOnce(profile);
    const dispatch = jest.fn();
    // criamos uma senha qualquer
    const password = 'senha qualquer';
    // disparamos o thunk fornecendo o usuário, senha
    // e a função dispatch
    await actions.login(profile.userName, password)(dispatch);
});
```

Em seguida, podemos fazer nossas asserções verificando se as chamadas à função `auth.logIn` foram feitas corretamente:

```
it('Deve disparar as ações de login com sucesso', async () => {
    auth.logIn.mockResolvedValueOnce(profile);
    const dispatch = jest.fn();
    const password = 'senha qualquer';
    await actions.login(profile.userName, password)(dispatch);

    // verificamos que auth.logIn foi chamada
    // uma única vez
    expect(auth.logIn).toHaveBeenCalledTimes(1);
    // e com os dados esperados de username/password
    expect(auth.logIn).toHaveBeenCalledWith({
      username: profile.userName,
      password
    });
});
```

E o mesmo para a função `dispatch`. Nesse cenário, acessar suas chamadas por `mock.call` deixa nosso teste bem mais simples, já que essa função foi chamada duas vezes:

```
it('Deve disparar as ações de login com sucesso', async () => {
    auth.logIn.mockResolvedValueOnce(profile);
    const dispatch = jest.fn();
    const password = 'senha qualquer';
    await actions.login(profile.userName, password)(dispatch);
```

```
    expect(auth.logIn).toHaveBeenCalledTimes(1);
    expect(auth.logIn).toHaveBeenCalledWith({
      username: profile.userName,
      password
    });

    // verificamos que dispatch foi chamada duas vezes
    expect(dispatch.mock.calls).toEqual([
      // uma para o início da autenticação
      [actions.authenticating()],
      // outra ao autenticar com sucesso
      [actions.authenticateSuccess(profile)]
    ]);
});
```

Bem prático, como qualquer outra função assíncrona que já testamos, certo?

Deixo como desafio para este momento a realização do teste para o cenário de erro desse `thunk`, assim como os testes para as funções `logout` e `checkAuthentication`, que são os outros dois `thunks` restantes, bem mais simples do que o que acabamos de testar.

Na sequência, vamos ver como testar os `reducers`, que produzem os estados da aplicação a partir dos valores anteriores e de novas ações.

Testando reducers

Testar os reducers será ainda mais fácil do que as ações, já que são funções puras (inclusive isso é um princípio do próprio Redux) e, basicamente, recebem um estado anterior e uma ação, produzindo um novo estado como saída.

Reducers são baseados nesses estados e ações por padrão, possuindo uma estrutura de `switch/case` dentro de sua função. Dentro de cada `case` um novo estado é retornado. Simples assim.

Vamos criar nosso arquivo `reducer.unit.js` dentro da pasta `src/__tests__/store/user` e importar nosso `reducer`, que é exportado como `default`. Além disso, os arquivos dos reducers desse projeto também exportam uma variável `INITIAL_STATE` com um estado inicial com os dados da store vazios. Como pode ser algo interessante para nosso teste, vamos importar essa variável também:

```
// arquivo src/__tests__/store/user/reducer.unit.js
// importamos o reducer e o estado inicial
import reducer, { INITIAL_STATE } from '../../../store/user/reducer';
```

Como o `reducer` muda um campo `status`, usado para indicar como está a ação de login dentro desse estado global de usuário, podemos importar a variável `LOADING_STATUS`. Como vamos precisar disparar ações contra esse `reducer`, vamos importar tudo das nossas actions:

```
import reducer, { INITIAL_STATE } from '../../../store/user/reducer';
// importamos todos os valores exportados das actions
import * as actions from '../../../store/user/actions';
```

Agora vamos criar nosso bloco `describe` / `it` para testarmos a primeira ação, a de `AUTHENTICATE_PENDING`:

```
import reducer, { INITIAL_STATE } from '../../../store/user/reducer';
import * as actions from '../../../store/user/actions';

// criamos bloco describe
describe('Reducer de usuário', () => {
  // criamos bloco it
  it('AUTHENTICATE_PENDING', () => {
  });
});
```

Essa primeira ação basicamente muda o valor de `status` no estado. Vamos disparar nosso `reducer` fornecendo o `INITIAL_STATE` como primeiro argumento e, como segundo, o retorno da função que cria essa ação (`actions.authenticating`):

```
describe('Reducer de usuário', () => {
  it('AUTHENTICATE_PENDING', () => {
    // executamos o reducer fornecendo o estado inicial
    // e o retorno do action creator que inicia a autenticação
    const state = reducer(INITIAL_STATE, actions.authenticating());
  });
});
```

Em vez de executar `actions.authenticating()`, poderíamos também passar o objeto com o `type` correto diretamente, sem problemas.

Como esse cenário apenas muda o `status` do estado global relacionado aos dados de usuário, podemos verificar se o valor de `state.status` é igual ao valor presente em `LOADING_STATUS.LOADING`, exportado pelas actions:

```
describe('Reducer de usuário', () => {
  it('AUTHENTICATE_PENDING', () => {
    const state = reducer(INITIAL_STATE, actions.authenticating());

    // verificamos que state.status
    // é igual a actions.LOADING_STATUS.LOADING
    expect(state.status).toEqual(actions.LOADING_STATUS.LOADING);
  });
});
```

Para o próximo teste, vamos criar nosso `it` para a action de `AUTHENTICATE_SUCCESS` quando o usuário é administrador, dentro desse mesmo `describe`:

```
it(`AUTHENTICATE_PENDING quando o usuário é admin`, () => {
});
```

Para esse cenário, precisaremos disparar a action resultante da função `actions.authenticateSuccess` com os dados de um usuário. Vamos importar nosso mock, no início do arquivo:

```
import { profile } from '../../../mocks/profile';
```

Dentro do nosso teste, vamos criar uma variável `dados`, que receberá todos os dados de perfil, além de sobrescrever o valor de

`role` para `ADMIN`:

```
it(`AUTHENTICATE_PENDING quando o usuário é admin`, () => {
    const dados = {
      ...profile,
      role: 'ADMIN'
    };
});
```

Podemos executar nosso `reducer` com o estado inicial, fornecendo como `action` o valor correto:

```
it(`AUTHENTICATE_PENDING quando o usuário é admin`, () => {
    const dados = {
      ...profile,
      role: 'ADMIN'
    };
    // executamos o reducer fornecendo o estado inicial
    // e o retorn de actions.authenticateSuccess com os dados criados
    const state = reducer(INITIAL_STATE,
actions.authenticateSuccess(dados));
});
```

Agora, podemos realizar as asserções para esse cenário. Podemos verificar que:

- O `status` desse novo estado deve ser igual ao valor de `LOADING_STATUS.LOADED`;
- O valor de `info` deve ser os dados do usuário logado;
- O valor de `authenticated` deve ser `true`;
- O valor de `admin` deve ser `true` também:

```
it(`AUTHENTICATE_PENDING quando o usuário é admin`, () => {
    const dados = {
      ...profile,
      role: 'ADMIN'
    };
    const state = reducer(INITIAL_STATE,
actions.authenticateSuccess(dados));

    // verificamos se status é igual a LOADING_STATUS.LOADED
```

```
    expect(state.status).toEqual(actions.LOADING_STATUS.LOADED);
    // verificamos se o valor de info é igual aos dados
    expect(state.info).toEqual(dados);
    // verificamos se o valor de authenticated é true
    expect(state.authenticated).toEqual(true);
    // verificamos se o valor de admin é true
    expect(state.admin).toEqual(true);
});
```

E para que possamos testar o cenário em que o usuário não é admin, podemos copiar todo esse teste e fazer algumas pequenas alterações, removendo a `role` da variável `dados` (nem precisamos mais dela, na verdade) e também modificando a última asserção, já que o usuário não será admin:

```
it(`AUTHENTICATE_PENDING quando o usuário não é admin`, () => {
    // apagamos a variável dados
    // e fornecemos o profile diretamente
    const state = reducer(INITIAL_STATE,
actions.authenticateSuccess(profile));

    expect(state.status).toEqual(actions.LOADING_STATUS.LOADED);
    // alteramos aqui para utilizar profile ao invés de dados
    expect(state.info).toEqual(profile);
    expect(state.authenticated).toEqual(true);
    // verificamos que o admin não possui o valor true
    expect(state.admin).toEqual(false);
});
```

Já vamos testar nosso penúltimo case, o que realiza o logout do usuário. Vamos criar mais um `it` :

```
it('LOGOUT apaga os dados armazenados sobre o usuário autenticado', () =>
{
});
```

Para que esse teste seja mais próximo ao cenário que ocorre na aplicação, vamos criar um estado onde já existe uma pessoa autenticada. Podemos usar como base qualquer um dos dois últimos cenários de teste:

```
it('LOGOUT apaga os dados armazenados sobre o usuário autenticado', () =>
{
    // executamos o reducer para o cenário de autenticação
    // do usuário de mock
    const authenticated = reducer(INITIAL_STATE,
actions.authenticateSuccess(profile));
});
```

E vamos disparar mais uma vez o `reducer`, mas, dessa vez, o primeiro argumento (o estado) será a variável que acabamos de criar e o segundo será a action que realiza o logout (`clearAuthentication`). Simularemos, dessa forma, um usuário autenticado realizando seu logout:

```
it('LOGOUT apaga os dados armazenados sobre o usuário autenticado', () =>
{
    const authenticated = reducer(INITIAL_STATE,
actions.authenticateSuccess(profile));
    // executamos o reducer com o estado autenticado
    // e com a ação de limpar a autenticação
    const state = reducer(authenticated, actions.clearAuthentication())
});
```

Com esse novo estado, podemos verificar que:

- O valor de `authenticated` agora é `false`;
- O valor de `admin` agora é `null`;
- O valor de `info` é um objeto vazio `{}`.

```
it('LOGOUT apaga os dados armazenados sobre o usuário autenticado', () =>
{
    const authenticated = reducer(INITIAL_STATE,
actions.authenticateSuccess(profile));
    const state = reducer(authenticated, actions.clearAuthentication())

    // verificamos o valor de authenticated
    expect(state.authenticated).toEqual(false);
    // verificamos o valor de admin
    expect(state.admin).toEqual(null);
    // verificamos o valor de info
```

```
    expect(state.info).toEqual({});
});
```

Por último, mas não menos importante, existe um `case default` no `switch` dos `reducers` no qual, caso uma ação não mapeada seja disparada, apenas retorna o estado anterior. Vamos criar um teste bem simples para este cenário, disparando o `INITIAL_STATE` como primeiro argumento e, como segundo, um objeto com um `type` qualquer:

```
// criamos um novo it
it('Retorna o estado recebido caso a ação não seja mapeada', () => {
    // executamos o reducer com o INITIAL_STATE
    // e uma ação qualquer, não mapeada
    const state = reducer(INITIAL_STATE, { type: 'ação não mapeada' });

    // verificamos que o state retornado é igual ao INITIAL_STATE
    expect(state).toEqual(INITIAL_STATE);
});
```

É um cenário simples, mas importante de ser testado, já que o padrão de um `reducer` é retornar os dados recebidos caso não possua nenhum `case` para tratar uma action.

Com isso, o desafio dessa vez é realizar os testes dos dados restantes da *store*: os relacionados às notificações (dentro da pasta `notification`) e os relacionados aos perfis dos usuários (dentro da pasta `profiles`). O fluxo será exatamente como fizemos até aqui, testando as `actions`, os `action creators`, os `thunks` e seus devidos `reducers`.

11.5 Próxima parada: testes de integração no front-end

Mais uma etapa concluída na nossa jornada de entregar código com qualidade. Vimos bastante coisa neste capítulo!

De componentes simples de interface, aos utilitários de cliente, até a estrutura de estado global da nossa aplicação. O tempo voa quando a gente se diverte, não é?

No próximo capítulo, veremos como testar de forma integrada os diversos componentes de nossa aplicação, testando suas páginas por completo.

Vamos lá!

Capítulo 12
Testes de integração nas telas da aplicação

Se pensarmos que, fundamentalmente, um teste de integração testa vários módulos em conjunto, quando testamos a aplicação back-end tivemos o trabalho de executar um servidor e disparar requisições simulando a utilização da API.

Para os cenários de interface, já testamos os componentes de forma isolada. Agora a lógica será a mesma: testar se os conjuntos de nossa interface trabalham de forma coerente em conjunto. Poderíamos fazer isso renderizando toda a nossa aplicação e garantindo todos os fluxos de cadastro e edição, sem problemas. No entanto, esse trabalho completo fica mais garantido quando falamos de teste end-to-end, que veremos mais para a frente.

Para esses próximos testes, vamos pensar na integração como sendo cada uma das páginas (que conterá diversos componentes dentro de si) e seu funcionamento específico, o que funciona perfeitamente para a nossa aplicação.

Dessa forma, o intuito dos testes de integração no front-end é renderizar as páginas de nossa aplicação e testar o funcionamento dos componentes de forma integrada. Podemos pensar, em resumo, que os comportamentos de cada página são os seguintes:

- A página de login deve renderizar um formulário com um card e um título coerente e disparar as devidas requisições para tratar os dados de algum usuário e tentar realizar uma autenticação;
- A página de dashboard deve permitir abrir/fechar a modal de cadastro e de edição de usuário (somente para usuários administradores) assim como fazer as determinadas requisições para manipular esses perfis.

Dessa forma, podemos realizar mocks somente das nossas requisições e garantir que toda a aplicação está trabalhando como o

esperado.

12.1 Testando a página de Dashboard

Vamos realizar os testes da página de dashboard de forma integrada. Dentro da nossa pasta `src/__tests__` , vamos criar a pasta `pages` e, dentro dela, o arquivo `dashboard.integration.js` . Iniciaremos importando os módulos já conhecidos de `screen` , `userEvent` e, desta vez, vamos utilizar a função `renderWithProviders` para simular o Redux. Vamos também importar o componente da tela:

```
// arquivo src/__tests__/pages/dashboard.integration.js
// importamos screen
import { screen, waitFor } from '@testing-library/react';
// importamos userEvent
import userEvent from '@testing-library/user-event';
// importamos função que renderizará a página
import { renderWithProviders } from '../../testUtils';
// importamos o componente
import DashboardPage from '../../pages/dashboard';
```

Para que as requisições à API não sejam feitas, precisaremos realizar alguns mocks e tratar alguns dados de retorno. Vamos importar `profile` para simular um usuário no sistema e `profileList` para simular uma lista de usuários. Também vamos importar e realizar o mock dos clientes de `profiles` e `authentication` :

```
// Utilitários
import { screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import { renderWithProviders } from '../../testUtils';

// Página
import DashboardPage from '../../pages/dashboard';

// Mocks
// importamos profile e profileList
```

```
import { profile, profileList } from '../../mocks/profile';
// importamos clientes de profile e authentication
import * as client from '../../clients/http/profiles';
import * as auth from '../../clients/http/authentication';
// realizamos o mock dos clientes
jest.mock('../../clients/http/profiles');
jest.mock('../../clients/http/authentication');
```

Testando como usuário comum

Começaremos simulando um usuário comum, isto é, um perfil com a `role` de `USER`. Primeiro, vamos criar alguns blocos `describe`:

```
describe('<DashboardPage />', () => {
  describe('Role USER', () => {});
});
```

Agora, vamos fazer algumas configurações no hook `beforeEach` de cada um desses `describes`. Como precisaremos sempre retornar uma lista de usuários da nossa API e também simular um usuário logado, vamos fazer mock dos valores de `client.getProfiles` e `auth.isLoggedIn` dentro do primeiro `describe`:

```
describe('<DashboardPage />', () => {
  // criamos beforeEach
  beforeEach(() => {
    // mockando retorno de getProfiles
    client.getProfiles.mockResolvedValue(profileList);
    // e retorno de isLoggedIn
    auth.isLoggedIn.mockReturnValue(true);
  });

  describe('Role USER', () => {});
```

Dentro do segundo `describe`, onde vamos realizar os testes de usuários que não são admin, vamos colocar um `beforeEach` para que todos os testes se autentiquem com o nosso perfil de mock. Podemos fazer isso fazendo um mock do retorno da função `auth.getLoggedUser`:

```
describe('<DashboardPage />', () => {
  beforeEach(() => {
    client.getProfiles.mockResolvedValue(profileList);
    auth.isLoggedIn.mockReturnValue(true);
  });

  describe('Role USER', () => {
    // inserimos beforeEach
    beforeEach(() => {
      // simulamos um usuário autenticado
      auth.getLoggedUser.mockReturnValueOnce(profile);
    });
  });
});
```

Vamos começar por um teste simples, verificando se a lista de perfis é exibida. Crie um `it` para isso:

```
it('Deve renderizar completamente com o Carrossel de perfis', async () =>
{});
```

Dentro desse teste precisamos renderizar nossa página completa com a função `renderWithProviders` :

```
it('Deve renderizar completamente com o Carrossel de perfis', async () =>
{
    // renderizamos a tela com os providers
    renderWithProviders(<DashboardPage />);
});
```

Já podemos consultar pela role `listitem` todos os itens do carrossel. Ao renderizar a tela, ocorre uma sequência de ações (como a requisição que consulta os dados dos perfis e checa se existe uma pessoa autenticada) que disparam algumas re-renderizações na aplicação. Nesse cenário, podemos utilizar a consulta com `find*` , para que ela seja uma Promise que é resolvida com os dados dos usuários ao final de todo esse processo:

```
it('Deve renderizar completamente com o Carrossel de perfis', async () =>
{
    renderWithProviders(<DashboardPage />);
```

```
    // consultamos todos os itens do tipo listitem
    const items = await screen.findAllByRole('listitem');
});
```

Em seguida, podemos realizar uma asserção, verificando que a quantidade de perfis no carrossel é igual à quantidade de itens no mock da lista de perfil:

```
it('Deve renderizar completamente com o Carrossel de perfis', async () =>
{
    renderWithProviders(<DashboardPage />);

    const items = await screen.findAllByRole('listitem');

    // asserção que valida se a quantidade de itens em tela
    // é igual à quantidade de itens na lista de mock
    expect(items.length).toEqual(profileList.length);
});
```

Pronto, já temos o nosso primeiro teste de integração!

Para o próximo teste, sabemos que existe uma restrição quando o perfil de um usuário é do tipo `USER`. Essa restrição indica que um usuário desse tipo não pode realizar cadastro, edição ou remoção de perfis. Podemos verificar em nosso teste se esses respectivos botões estão desabilitados e se, ao clicar neles, as respectivas modais não aparecem.

Vamos começar com mais um `it` dentro de todos esses `describes` e também já vamos renderizar a tela da nossa aplicação:

```
// criamos o it
it('Não deve permitir cadastro/edição/remoção de perfil', async () => {
    // renderizando a aplicação
    renderWithProviders(<DashboardPage />);
});
```

Agora, podemos verificar se a modal (pela role `dialog`) não está na tela através da asserção `.not.toBeInTheDocument()`:

```
it('Não deve permitir cadastro/edição/remoção de perfil', async () => {
    renderWithProviders(<DashboardPage />);
    // consultamos a modal pela role dialog
    const modal = screen.queryByRole('dialog');
    // verificamos que não está na tela
    expect(modal).not.toBeInTheDocument();
});
```

Em seguida, vamos consultar um botão com o label `cadastrar` e garantir que ele está desabilitado usando a asserção `toBeDisabled()`:

```
it('Não deve permitir cadastro/edição/remoção de perfil', async () => {
    renderWithProviders(<DashboardPage />);

    const modal = screen.queryByRole('dialog');
    expect(modal).not.toBeInTheDocument();

    // consultamos o botão de cadastrar usuário
    const createButton = await screen.findByLabelText('cadastrar');
    // verificamos que está desabilitado
    expect(createButton).toBeDisabled();
});
```

Se quisermos ter ainda mais certeza de que a ação de clique não abre a modal, poderíamos fazer mais uma asserção como:

```
userEvent.click(createButton);
expect(modal).not.toBeInTheDocument();
```

Mas, para nosso cenário, a validação de desabilitado já é o suficiente.

Vamos fazer a mesma verificação com o botão de editar através de seu label `editar`. Como existem vários desses botões (um para cada perfil), podemos utilizar a função `findAll*` e pegar somente o primeiro botão, já que esse funcionamento é replicado para os demais. Depois, podemos repetir a mesma asserção com esse botão:

```
it('Não deve permitir cadastro/edição/remoção de perfil', async () => {
    renderWithProviders(<DashboardPage />);
```

```
    const modal = screen.queryByRole('dialog');
    expect(modal).not.toBeInTheDocument();

    const createButton = await screen.findByLabelText('cadastrar');
    expect(createButton).toBeDisabled();
    // consultamos os botões de editar e selecionamos o primeiro
    const [editButton] = await screen.findAllByLabelText('editar');
    // verificamos que está desabilitado
    expect(editButton).toBeDisabled();
});
```

Por fim, podemos fazer o mesmo processo para o botão de deletar, selecionando seu label `deletar` e assegurando que está desabilitado. Como também existe um para cada perfil, também podemos consultar todos e selecionar somente o primeiro:

```
it('Não deve permitir cadastro/edição/remoção de perfil', async () => {
    renderWithProviders(<DashboardPage />);

    const modal = screen.queryByRole('dialog');
    expect(modal).not.toBeInTheDocument();

    const createButton = await screen.findByLabelText('cadastrar');
    expect(createButton).toBeDisabled();

    const [editButton] = await screen.findAllByLabelText('editar');
    expect(editButton).toBeDisabled();
    // consultamos os botões de deletar e selecionamos o primeiro
    const [deleteButton] = await screen.findAllByLabelText('deletar');
    // verificamos que está desabilitado
    expect(deleteButton).toBeDisabled();
});
```

Vamos prosseguir com os testes caso o usuário autenticado possua perfil de administrador.

Testando como administrador

Para começar, vamos criar mais um `describe` para essa categoria, no mesmo nível desse último que criamos. Nossa estrutura deve ser algo como:

```
describe('<DashboardPage />', () => {
  beforeEach(() => {
    client.getProfiles.mockResolvedValue(profileList);
    auth.isLoggedIn.mockReturnValue(true);
  });

  describe('Role USER', () => {
    // testes que acabamos de fazer
  });

  // novo describe
  describe('Role ADMIN', () => {});
});
```

Realizando mock dos dados de um usuário logado

Dentro dele, precisaremos simular um perfil do tipo `ADMIN`. Vamos fazer isso simulando o retorno da função `getLoggedUser` para possuir uma `role` de `ADMIN`, e vamos deixar isso em mais um `beforeEach`, somente dentro desse `describe`:

```
describe('Role ADMIN', () => {
    // beforeEach
    beforeEach(() => {
      // simula o retorno de getLoggedUser
      // com o perfil de mock
      // mas sobrescrevendo a role para ser ADMIN
      auth.getLoggedUser.mockReturnValueOnce({ ...profile, role: 'ADMIN'
});
    });
});
```

Agora, assim como o cenário de usuário do tipo `USER`, podemos verificar que, ao acessar a tela, devem ser carregados os perfis no carrossel. Podemos copiar o primeiro teste do `describe` anterior:

```
// copiamos o teste do describe anterior
it('Deve renderizar completamente com o Carrossel de perfis', async () =>
{
    renderWithProviders(<DashboardPage />);

    const items = await screen.findAllByRole('listitem');
    expect(items.length).toEqual(profileList.length);
});
```

Testando modal de cadastro de perfil

Entretanto, diferente de um usuário comum, administradores podem alterar registros dos perfis de usuário. Para que possamos separar as ações, vamos realizar um teste que apenas verifica a abertura/fechamento da modal de cadastro. Começaremos com mais um `it`:

```
it('Abre/fecha a modal de cadastro de perfil', async () => {};
```

Vamos renderizar a nossa tela e, após isso, faremos o processo inverso que fizemos nos testes anteriores, validando que o botão de cadastrar usuário não está desabilitado:

```
it('Abre/fecha a modal de cadastro de perfil', async () => {
    // renderizamos a tela
    renderWithProviders(<DashboardPage />);
    // consultamos o botão pelo label cadastrar
    const createButton = await screen.findByLabelText('cadastrar');
    // verificamos que não está desabilitado
    expect(createButton).not.toBeDisabled();
});
```

Podemos clicar nesse botão e verificar se a modal estará presente na nossa tela. Vamos aproveitar e verificar que o elemento com o texto "Criar dados de perfil" (que é o título da modal) também existe:

```
it('Abre/fecha a modal de cadastro de perfil', async () => {
    renderWithProviders(<DashboardPage />);
```

```
    const createButton = await screen.findByLabelText('cadastrar');
    expect(createButton).not.toBeDisabled();
    // disparamos um clique
    userEvent.click(createButton);
    // verificamos se a modal agora está na tela
    expect(screen.queryByRole('dialog')).toBeInTheDocument();
    // verificamos se o elemento de título existe
    expect(screen.queryByText('Criar dados de
Perfil')).toBeInTheDocument();
});
```

Da mesma forma que testamos a abertura, podemos validar o fechamento da modal. Podemos consultar o botão com o label `fechar` e disparar um clique nele, também verificando se a modal sairá da tela após esse processo:

```
it('Abre/fecha a modal de cadastro de perfil', async () => {
    renderWithProviders(<DashboardPage />);

    const createButton = await screen.findByLabelText('cadastrar');
    expect(createButton).not.toBeDisabled();

    userEvent.click(createButton);
    expect(screen.queryByRole('dialog')).toBeInTheDocument();
    // consultamos o botão fechar por seu label
    const fechar = screen.getByLabelText('fechar');
    // clicamos nele
    userEvent.click(fechar);
    // verificamos que agora a modal não está na tela
    expect(screen.queryByRole('dialog')).not.toBeInTheDocument();
    // e que o título não está mais presente também
    expect(screen.queryByText('Criar dados de
Perfil')).not.toBeInTheDocument();
});
```

E pronto, verificamos o funcionamento de aparição/remoção da modal de cadastro. Ainda testaremos o fluxo de cadastro completo em breve.

Testando modal de edição de perfil

Podemos duplicar esse teste que acabamos de realizar para testar a aparição da modal de edição fazendo alguns pequenos ajustes, como:

- Título do teste deve ser diferente;
- Botão que dispara a modal deve ser um botão de editar;
- Título da modal.

Vamos lá:

```
// copiamos o teste modificando o título na parte "modal de edição"
it('Abre/fecha a modal de edição de perfil', async () => {
    renderWithProviders(<DashboardPage />);

    // consultamos todos os botões de editar
    // mas selecionamos o primeiro
    const [editButton] = await screen.findAllByLabelText('editar');
    // asseguramos que não está desabilitado
    expect(editButton).not.toBeDisabled();
    // clicamos no botão
    userEvent.click(editButton);
    // asseguramos que a modal está na tela
    expect(screen.queryByRole('dialog')).toBeInTheDocument();
    // com o título correto de editar
    expect(screen.queryByText('Editar dados de
Perfil')).toBeInTheDocument();
    // consultamos o botão que fecha a modal
    const fechar = screen.getByLabelText('fechar');
    // e disparamos um clique nele
    userEvent.click(fechar);
    // também assegurando que a modal sairá da tela
    expect(screen.queryByRole('dialog')).not.toBeInTheDocument();
    // e que o título também sairá
    expect(screen.queryByText('Editar dados de
Perfil')).not.toBeInTheDocument();
});
```

Perfeito! Com isso, temos dois testes que asseguram a aparição/remoção da modal de cadastro e edição.

Testando o cadastro de um perfil

Para que possamos testar um fluxo de cadastro, vamos criar mais um `it`:

```
it('Deve permitir o cadastro de perfil', async () => {});
```

Antes de renderizar a aplicação, precisaremos realizar um mock de um usuário para cadastrá-lo, assim como o retorno da resposta da API. Vamos criar uma nova variável que utilizará os dados de mock da variável `profile` e sobrescreverá com alguns novos valores:

```
it('Deve permitir o cadastro de perfil', async () => {
  // criamos variável newProfile
  // sobrescrevendo userName, email e password
  const newProfile = {
    ...profile,
    userName: 'username-qualquer',
    email: 'email@qualquer.com',
    password: 'senha secreta'
  };
});
```

Agora vamos indicar que esses dados serão a resposta da API, acessada pela função `client.createProfile`:

```
it('Deve permitir o cadastro de perfil', async () => {
  const newProfile = {
    ...profile,
    userName: 'username-qualquer',
    email: 'email@qualquer.com',
    password: 'senha secreta'
  };
  // mockamos a resolução da Promise que faz a chamada
  // com a variável que criamos
  client.createProfile.mockResolvedValueOnce(newProfile);
});
```

Com tudo configurado, podemos renderizar nossa tela:

```
it('Deve permitir o cadastro de perfil', async () => {
  const newProfile = {
    ...profile,
    userName: 'username-qualquer',
    email: 'email@qualquer.com',
    password: 'senha secreta'
  };
  client.createProfile.mockResolvedValueOnce(newProfile);
  // renderizamos a tela
  renderWithProviders(<DashboardPage />);
});
```

E clicar no botão cadastrar, para que possamos preencher a modal de cadastro:

```
it('Deve permitir o cadastro de perfil', async () => {
  const newProfile = {
    ...profile,
    userName: 'username-qualquer',
    email: 'email@qualquer.com',
    password: 'senha secreta'
  };
  client.createProfile.mockResolvedValueOnce(newProfile);
  renderWithProviders(<DashboardPage />);
  // consultamos o botão de cadastro
  const createButton = await screen.findByLabelText('cadastrar');
  // disparamos um clique
  userEvent.click(createButton);
});
```

Preenchendo dados do formulário Em seguida, precisaremos preencher os dados do formulário, sendo eles:

- O campo de e-mail;
- O campo de usuário;
- O campo de senha;
- O campo de nome;
- O campo de sobrenome;

- O campo de role.

Podemos fazer isso com as funções de `userEvent`, que já conhecemos, sendo a `userEvent.type` para simular a digitação em um campo e `selectOptions` para selecionar uma opção. Podemos preencher todos os campos com os dados da variável `newProfile`:

```
it('Deve permitir o cadastro de perfil', async () => {
  const newProfile = {
    ...profile,
    userName: 'username-qualquer',
    email: 'email@qualquer.com',
    password: 'senha secreta'
  };
  client.createProfile.mockResolvedValueOnce(newProfile);
  renderWithProviders(<DashboardPage />);

  const createButton = await screen.findByLabelText('cadastrar');
  userEvent.click(createButton);

  // preenchemos o campo de email
  userEvent.type(screen.getByPlaceholderText('email'), newProfile.email);
  // preenchemos o campo de usuario
  userEvent.type(screen.getByPlaceholderText('usuario'),
newProfile.userName);
  // preenchemos o campo de senha
  userEvent.type(screen.getByPlaceholderText('sua senha super secreta'),
newProfile.password);
  // preenchemos o campo de nome
  userEvent.type(screen.getByPlaceholderText('nome'), newProfile.name);
  // preenchemos o campo de sobrenome
  userEvent.type(screen.getByPlaceholderText('sobrenome'),
newProfile.lastName);
  // selecionamos a opção de role de USER
  userEvent.selectOptions(screen.getByPlaceholderText('role'),
screen.getByText('Usuário'));
});
```

Agora, só precisamos disparar um clique no botão de confirmar:

```
it('Deve permitir o cadastro de perfil', async () => {
  const newProfile = {
    ...profile,
    userName: 'username-qualquer',
    email: 'email@qualquer.com',
    password: 'senha secreta'
  };
  client.createProfile.mockResolvedValueOnce(newProfile);
  renderWithProviders(<DashboardPage />);

  const createButton = await screen.findByLabelText('cadastrar');
  userEvent.click(createButton);

  userEvent.type(screen.getByPlaceholderText('email'), newProfile.email);
  userEvent.type(screen.getByPlaceholderText('usuario'),
newProfile.userName);
  userEvent.type(screen.getByPlaceholderText('sua senha super secreta'),
newProfile.password);
  userEvent.type(screen.getByPlaceholderText('nome'), newProfile.name);
  userEvent.type(screen.getByPlaceholderText('sobrenome'),
newProfile.lastName);
  userEvent.selectOptions(screen.getByPlaceholderText('role'),
screen.getByText('Usuário'));
  // clicamos no botão confirmar
  userEvent.click(screen.getByText('Confirmar'));
});
```

Como isso vai realizar uma chamada à API (que *mockamos* no início do teste), que, por consequência, vai re-renderizar a tela da aplicação, podemos utilizar o utilitário `waitFor` para realizar nossas consultas somente após todas essas atualizações.

Dentro dele, podemos consultar todos os itens do carrossel e realizar uma asserção de que a quantidade de itens será igual à quantidade de itens em `profileList` acrescido de `1`, já que acabamos de adicionar mais um perfil. Vamos lá:

```
it('Deve permitir o cadastro de perfil', async () => {
  const newProfile = {
    ...profile,
```

```
    userName: 'username-qualquer',
    email: 'email@qualquer.com',
    password: 'senha secreta'
  };
  client.createProfile.mockResolvedValueOnce(newProfile);
  renderWithProviders(<DashboardPage />);

  const createButton = await screen.findByLabelText('cadastrar');
  userEvent.click(createButton);

  userEvent.type(screen.getByPlaceholderText('email'), newProfile.email);
  userEvent.type(screen.getByPlaceholderText('usuario'),
newProfile.userName);
  userEvent.type(screen.getByPlaceholderText('sua senha super secreta'),
newProfile.password);
  userEvent.type(screen.getByPlaceholderText('nome'), newProfile.name);
  userEvent.type(screen.getByPlaceholderText('sobrenome'),
newProfile.lastName);
  userEvent.selectOptions(screen.getByPlaceholderText('role'),
screen.getByText('Usuário'));
  userEvent.click(screen.getByText('Confirmar'));

  // aguardamos até todas as re-renderizações acontecerem
  await waitFor(async () => {
    // consultamos todos os itens do carrossel
    const items = screen.queryAllByRole('listitem');
    // verificamos que a quantidade de itens
    // é igual à quantidade de itens em profileList + 1
    expect(items.length).toEqual(profileList.length + 1);
  });
});
```

Com isso, nosso teste já está bem bacana. Caso você queira dar um passo adiante, você pode verificar se o último item da lista possui os dados do usuário cadastrado corretamente também.

Outros utilitários da RTL (React Testing Library)

Vamos aproveitar para aprender um utilitário chamado `within` neste exemplo. Importe-o da `@testing-library/react` :

```
// importamos a função within
import { screen, waitFor, within } from '@testing-library/react';
```

Vamos voltar ao nosso teste e selecionar todos os elementos da lista. Vamos pegar o último elemento do carrossel e executar a função `within` nele. Isso nos retorna algumas funções para consultar elementos somente dentro (por isso, o nome `within`) de um elemento específico:

```
it('Deve permitir o cadastro de perfil', async () => {
  // restante do código omitido

  // selecionamos todos os itens
  const items = screen.queryAllByRole('listitem');
  // selecionamos o último
  const lastItem = items[items.length - 1];
  // executamos within esse último elemento
  const queries = within(lastItem);
});
```

Com isso, podemos fazer algumas asserções, verificando se esse último elemento possui os dados que registramos (como `userName` e `email`). Essas asserções podem ser feitas através da variável `queries` , que armazena o retorno da função `within` :

```
it('Deve permitir o cadastro de perfil', async () => {
  // restante do código omitido

  const items = screen.queryAllByRole('listitem');
  const lastItem = items[items.length - 1];
  const queries = within(lastItem);
  // consultamos se um texto com o userName está na tela
  expect(queries.getByText(newProfile.userName)).toBeInTheDocument();
  // consultamos se um texto com o email está na tela
  expect(queries.getByText(newProfile.email)).toBeInTheDocument();
});
```

Ainda podemos dar mais um passo antes de finalizar: garantir que o componente de Snackbar está sendo exibido como deveria.

Precisaremos ajustar nosso `beforeEach` (somente desse `describe`) para usar timers, e também inserir um `afterEach` para resetá-los ao valor real:

```
describe('Role ADMIN', () => {
  beforeEach(() => {
    auth.getLoggedUser.mockReturnValueOnce({ ...profile, role: 'ADMIN' });
    // ajustamos o beforeEach para usar temporizadores falsos
    jest.useFakeTimers();
  });
  afterEach(() => {
    // criamos um afterEach para resetar os temporizadores reais
    jest.useRealTimers();
  });
  // restante do código omitido
});
```

Agora, no teste, podemos avançar os timers pendentes e garantir que, após isso, a mensagem do Snackbar estará na tela. Como isso causa uma re-renderização em nossa aplicação, vamos aplicá-lo dentro de um `waitFor` também. Como não queremos validar mais nada relacionado aos timers, podemos executar a função `jest.runAllTimers()` após a asserção:

```
it('Deve permitir o cadastro de perfil', async () => {
  // restante do código omitido

  await waitFor(() => {
    jest.runOnlyPendingTimers();
    expect(screen.getByText('Perfil criado com
sucesso')).toBeInTheDocument();
    jest.runAllTimers();
  });
});
```

Dessa forma, conseguimos garantir que, além de cadastrar um usuário com sucesso, o feedback em tela também está correndo, apresenta o Snackbar de sucesso e insere o último perfil na lista.

Testando cenário de falha no cadastro

Podemos duplicar esse teste fazendo alguns pequenos ajustes para garantir o cenário de falha também. Precisaremos:

- Alterar o título do teste;
- Ajustar o mock do retorno da API para ser rejeitada com alguma mensagem;
- Validar que a quantidade de itens no carrossel não vai ser incrementada;
- Validar que o Snackbar mostrará a mensagem de erro em vez da mensagem de sucesso.

Vamos lá. Abaixo desse teste, podemos realizar o seguinte:

```
// alteramos o título do teste
it('Deve exibir um erro caso não consiga cadastrar um perfil', async () =>
{
  const newProfile = {
    ...profile,
    userName: 'username-qualquer',
    email: 'email@qualquer.com',
    password: 'senha secreta'
  };
  // criamos uma string de erro qualquer
  const error = 'Ocorreu um erro';
  // alteramos a Promise para ser rejeitada
  client.createProfile.mockRejectedValueOnce(error);
  renderWithProviders(<DashboardPage />);

  const createButton = await screen.findByLabelText('cadastrar');
  userEvent.click(createButton);

  userEvent.type(screen.getByPlaceholderText('email'), newProfile.email);
  userEvent.type(screen.getByPlaceholderText('usuario'),
newProfile.userName);
  userEvent.type(screen.getByPlaceholderText('sua senha super secreta'),
newProfile.password);
  userEvent.type(screen.getByPlaceholderText('nome'), newProfile.name);
  userEvent.type(screen.getByPlaceholderText('sobrenome'),
newProfile.lastName);
  userEvent.selectOptions(screen.getByPlaceholderText('role'),
```

```
screen.getByText('Usuário'));
  userEvent.click(screen.getByText('Confirmar'));

  await waitFor(() => {
    const items = screen.queryAllByRole('listitem');
    // garantimos que agora a quantidade de itens
    // permanece igual à quantidade de itens da lista
    expect(items.length).toEqual(profileList.length);
  });

  await waitFor(() => {
    jest.runOnlyPendingTimers();
    // validamos o Snackbar usando a mensagem de erro
    expect(screen.getByText('Ocorreu um erro ao criar o
perfil')).toBeInTheDocument();
    jest.runAllTimers();
  });
});
```

Para trabalhar com mensagens de erro, existe uma variável exportada pelo arquivo `src/store/notification/actions` chamada `MESSAGES`, que pode nos auxiliar a validar essas mensagens do Snackbar. Essa variável contém os valores de texto para todas as mensagens de sucesso e erro do Snackbar na aplicação. Vamos importá-la no início do nosso arquivo:

```
// importamos as mensagens do snackbar
import { MESSAGES } from '../../store/notification/actions';
```

Em seguida, onde inserimos os textos `Perfil criado com sucesso` e `Ocorreu um erro ao criar o perfil`, vamos trocar essas strings para `MESSAGES.CREATE.SUCCESS` e `MESSAGES.CREATE.ERROR`, que possuem esses mesmos valores respectivamente:

```
it('Deve permitir o cadastro de perfil', async () => {
  // restante do código omitido

  await waitFor(() => {
    jest.runOnlyPendingTimers();
    expect(screen.getByText(MESSAGES.CREATE.SUCCESS)).toBeInTheDocument();
```

```
    jest.runAllTimers();
  });
});

it('Deve exibir um erro caso não consiga cadastrar um perfil', async () =>
{
  // restante do código omitido

  await waitFor(() => {
    jest.runOnlyPendingTimers();
    expect(screen.getByText(MESSAGES.CREATE.ERROR)).toBeInTheDocument();
    jest.runAllTimers();
  });
});
```

Dessa forma, se alguma das mensagens alterarem, nosso teste não quebrará, já que validamos diretamente essa variável reutilizada.

Nossos testes para o cenário de criação de perfil já estão bem robustos! Conseguimos validar vários comportamentos e componentes integrados.

Se você notou, nosso processo para testar algo em integração até agora é basicamente o seguinte:

- Preparar os mocks necessários;
- Renderizar uma tela;
- Executar as ações necessárias;
- Validar se os acontecimentos ocorreram como o esperado.

Exatamente o mesmo processo dos testes unitários, só que em uma camada "superior", que organiza os componentes em conjunto.

Com isso, o desafio desta parte será a finalização dos testes dessa tela, restando o fluxo de:

- Atualização de usuário;
- Remoção de usuário.

Esses fluxos também podem possuir cenários de erro, mudando somente o resultado do retorno da API e a mensagem exibida no Snackbar. São caminhos que, com certeza, valem a pena serem testados como acabamos de fazer.

Após finalizar, você também pode realizar os testes de integração da tela de login, que são ainda mais simples.

12.2 Próxima parada: testes de regressão visual

Já garantimos o funcionamento de nossa aplicação de diversas maneiras. De forma unitária e também integrada. Mas ainda não testamos nada que validasse o aspecto visual de nossas mudanças.

No próximo capítulo, veremos como podemos rastrear e garantir que as mudanças visuais dos nossos componentes não passem despercebidas.

Capítulo 13
Testes de regressão visual

Até o momento nossos testes envolvem apenas as funcionalidades de nossas páginas. Quando se trata de aplicações front-end, mudanças visuais são críticas e devem ser levadas em consideração, pois podem facilmente impactar alguma ação do usuário.

Existem formas de garantir estilos de CSS nos testes unitários, mas esse trabalho fica muito mais claro e organizado utilizando ferramentas de regressão visual.

Antes de entender o que é um teste de regressão visual, vamos entender o que é um teste de regressão, por si só.

13.1 O que são testes de regressão

Um teste de regressão nada mais é do que uma categoria de testes que garante que, mediante novas alterações e funcionalidades em um sistema, o restante desse mesmo sistema não sofrerá nenhum impacto e continuará funcionando corretamente como esperado. Afinal, se você criou uma nova tela ou uma nova funcionalidade, o ideal é que o restante do seu software não seja impactado se nada mudou, certo? É para isso que servem testes de regressão.

Quando pensamos em regressão visual, aplicamos esse mesmo princípio para as nossas interfaces. Isso quer dizer que, conforme nossa aplicação cresce e novos componentes e interfaces surgem, garantimos que nenhum detalhe passa sem uma devida aprovação.

Como regressão visual funciona

Basicamente, o que uma ferramenta de regressão visual faz é tirar uma sequência de screenshots dos seus componentes e/ou das suas telas e salvar essas referências em alguma pasta (ou na nuvem, facilitando o processo) para validá-las futuramente.

Após gerar essas referências, a ideia é que a cada nova alteração no projeto (como uma nova funcionalidade), esse processo seja executado, gerando novos screenshots.

Dessa forma, a ferramenta consegue comparar as imagens da nova alteração com as de referência que possuía anteriormente. Aplicando alguns algoritmos de verificação e reconhecimento nos pixels em tela, a ferramenta verifica se houve mudança em algum componente de interface e permite que (caso algo tenha sido modificado) exista uma etapa extra de aprovação para que essas modificações sejam aprovadas antes de ir ao ar.

13.2 Conhecendo o Loki e executando os testes

Existem diversas ferramentas que nos auxiliam a realizar testes de regressão visual e que são configuradas de maneiras diferentes. A que vamos ver aqui se chama Loki (https://loki.js.org/).

Essa ferramenta utiliza como base os arquivos do Storybook (https://storybook.js.org/), aquele playground dos componentes que também já está configurado no projeto. Você já deve ter executado o comando `npm run storybook` quando começamos os testes unitários, mas, caso ainda não tenha feito, agora é mais uma oportunidade para você se familiarizar com esse "ambiente" de desenvolvimento de componentes extremamente útil!

Loki já está instalado no projeto e já existem alguns scripts que facilitam nossa vida. Inclusive, no arquivo `package.json`, está a configuração que indica os dispositivos que serão levados em consideração ao realizar os testes.

Para executar os testes de regressão visual e gerar os arquivos de referência, é necessário estar com o Storybook executando e, após isso, em outra aba do terminal, executar o comando `npm run test:regression:refs` (que executará `loki update`). Esse comando exibirá alguns logs no terminal ao longo de sua execução e, por fim, gerará os arquivos de screenshot de referência. Esses arquivos são gerados para cada tipo de story (do Storybook) configurado no componente.

Com isso, uma pasta `.loki` será criada na raiz da aplicação. Ao abri-la você pode perceber que algumas outras subpastas são criadas (como `current`, `difference`, `reference`). Dê uma olhada nelas e você conseguirá perceber que a pasta `reference` agora está recheada com as imagens de referência de nossos componentes.

Na sequência, vamos fazer uma pequena mudança. No componente `src/components/profile/index.js`, na linha `47`, vamos mudar o valor de `type="blue"` para `type="red"`. Isso fará com que o botão azul de editar no card de perfil seja alterado para vermelho.

Após isso, ainda com o Storybook executando, vamos executar o comando `npm run test:regression` (que vai executar `loki test`). Você verá que teremos algumas mensagens de erro, e os testes dos componentes de `Profile` e `Carousel` vão falhar com algumas mensagens parecidas com a seguinte:

```
FAIL  chrome.docker/chrome.laptop/Components/Carousel
    Editable
    Screenshot differs from reference, see
.loki/difference/chrome_laptop_Components_Carousel_Editable.png
FAIL  chrome.docker/chrome.laptop/Components/Profile
    Editable
    Screenshot differs from reference, see
.loki/difference/chrome_laptop_Components_Profile_Editable.png
```

Você deve perceber esse mesmo log para o teste que simula um celular (já que, geralmente, essas ferramentas também permitem que você simule um dispositivo móvel), tendo algo como:

```
FAIL  chrome.docker/chrome.iphone7/Components/Carousel
    Editable
    Screenshot differs from reference, see
.loki/difference/chrome_iphone7_Components_Carousel_Editable.png
FAIL  chrome.docker/chrome.iphone7/Components/Profile
    Editable
    Screenshot differs from reference, see
.loki/difference/chrome_iphone7_Components_Profile_Editable.png
```

O que muda dessa mensagem para a de cima é apenas o dispositivo testado: na primeira, o teste foi baseado em um navegador Chrome em um laptop (por isso `chrome.laptop` nas mensagens) e, na segunda, em um Chrome em iPhone7 (por isso `chrome.iphone7` nos logs).

O que essas mensagens indicam é que justamente os componentes de `Carousel` e `Profile` tiveram novos screenshots. Essas novas imagens, ao serem comparadas, foram diferentes das anteriores (de referência), por isso o erro "Screenshot differs from reference".

Vamos ver o screenshot do componente de Profile na pasta de referência:

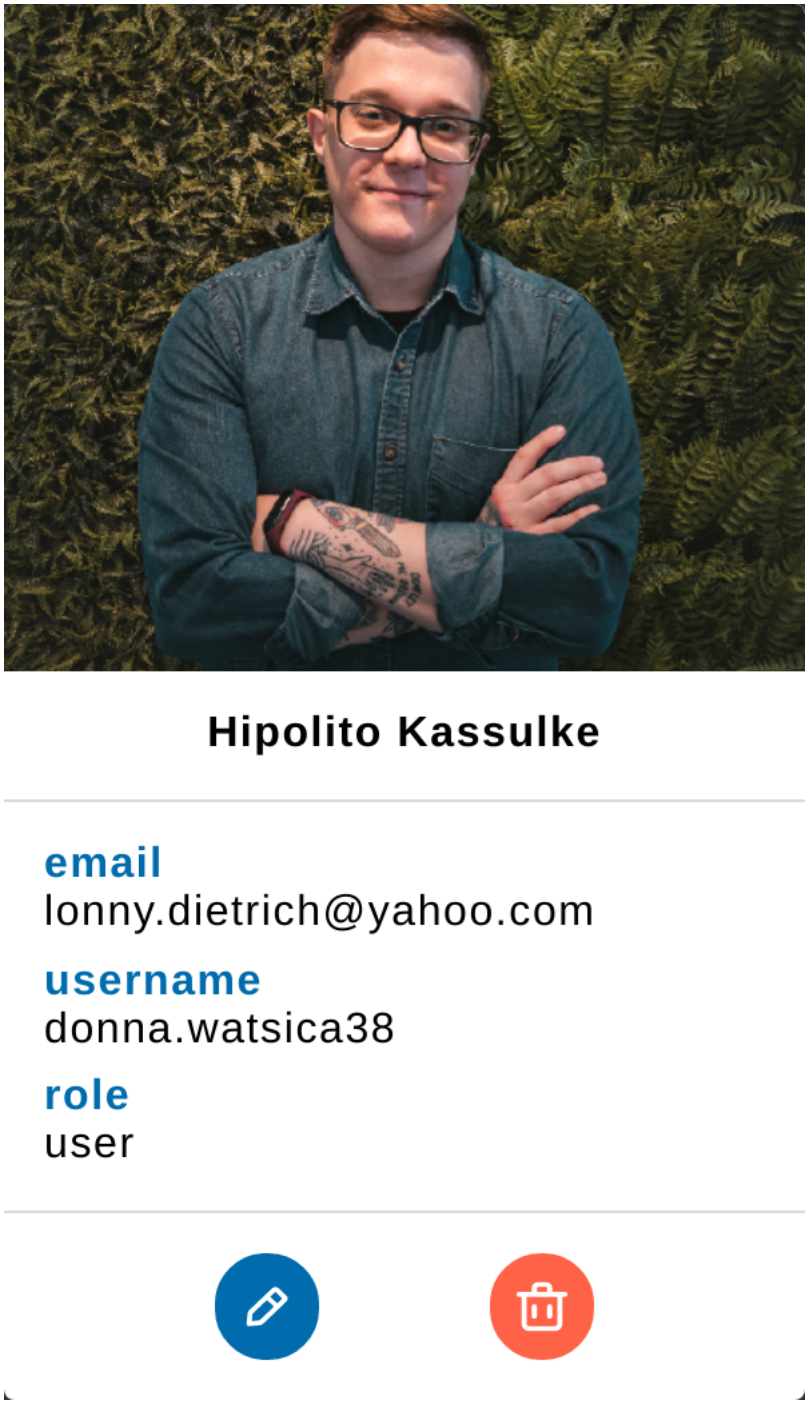


Figura 13.1: Componente de Profile de Referência.

Agora a pasta `.loki/difference` terá algumas imagens. Dê uma olhada nela e você verá que o conteúdo que alteramos estará preenchido em rosa, indicando a mudança de estilo nesse componente. Vamos ver o componente de Profile nesta pasta:

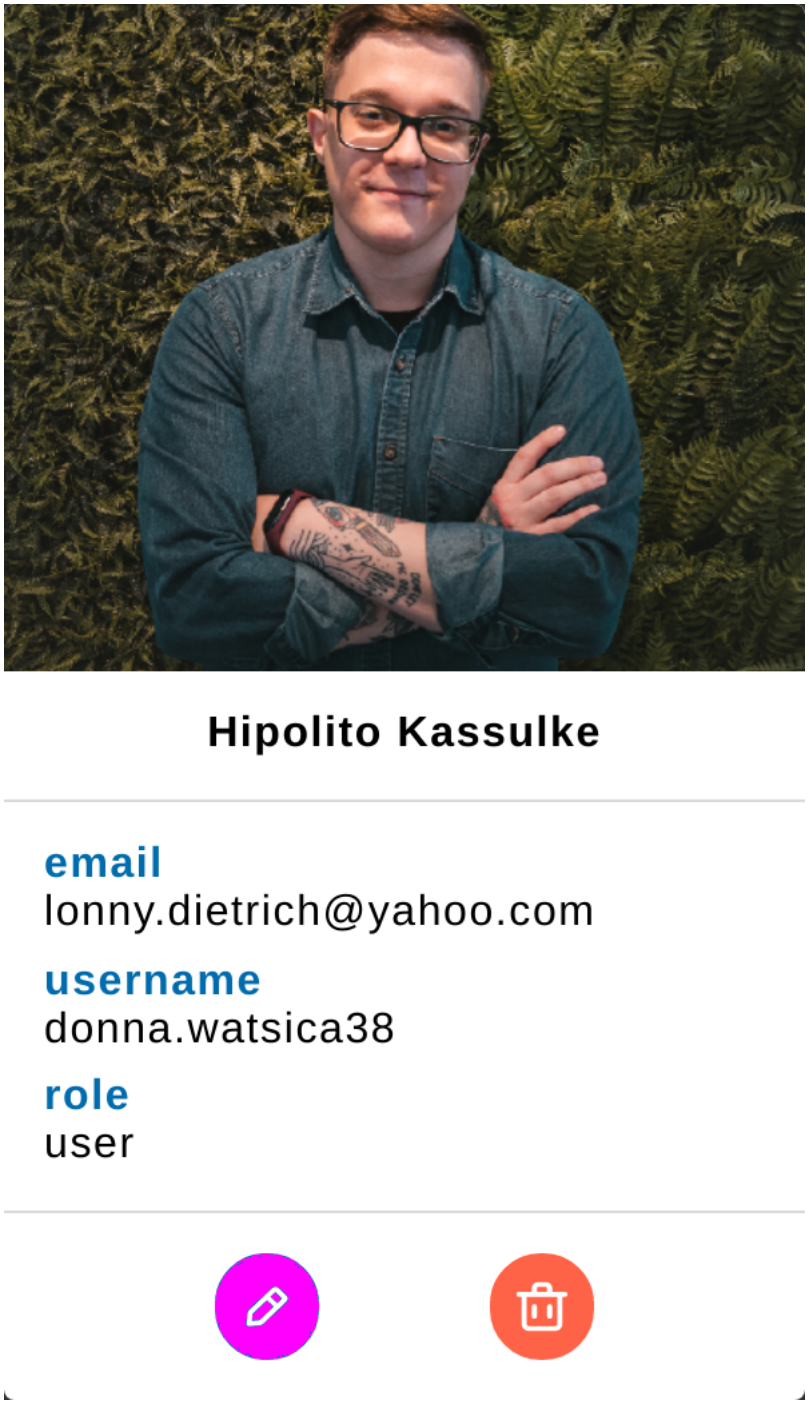


Figura 13.2: Componente de Profile de Diferença.

Percebeu a mudança de cor no botão que alteramos?

Caso fosse uma alteração necessária para a aplicação, seria possível aprová-la com o comando `npm run test:regression:approve`

(que executaria `loki approve`). Após sua execução, as imagens de referência seriam atualizadas, contendo esses últimos screenshots gerados.

Na última parte do livro, deixo uma lista com algumas outras ferramentas de regressão visual (assim como para os outros tipos de teste) para que você possa conhecer e analisar qual se encaixa melhor para a necessidade do seu projeto. Algumas ferramentas oferecem integrações mais interessantes para que esse processo de validação seja automatizado com o mínimo de esforço possível e possua indicativos de sucesso/erro de forma mais clara.

Geralmente esses testes demandam uma certa configuração e infraestrutura, já que é necessário automatizar essa tarefa de validação para gerar os screenshots, e isso pode mudar bastante dependendo das necessidades, de como seu projeto foi desenvolvido e de como seu CI/CD está funcionando. Contudo, o que aprendemos até aqui já é o suficiente para que você entenda fundamentalmente como funciona um teste de regressão visual para que possa aplicá-lo em conjunto com sua infraestrutura específica, escolhendo a ferramenta que mais se adequar ao seu propósito.

13.3 Grand Finale: hora de testar o funcionamento completo de nosso software

Até agora testamos as camadas do nosso sistema de forma unitária e integrada, mas completamente isoladas (front-end e back-end).

Antes de finalizarmos nossa jornada, precisamos dar mais um passo: testar o funcionamento completo do nosso sistema, de ponta a ponta (ou end-to-end). Vamos lá!

Parte 5: Testando de ponta a ponta

Agora que temos todas as nossas camadas (front-end e back-end) devidamente testadas, chegou a hora de garantir que todo o nosso fluxo funciona de ponta a ponta (ou *end-to-end*/e2e).

Capítulo 14
Testes de ponta a ponta (end-to-end)

Chegou a hora de testar todo o funcionamento de nossa aplicação, front-end e back-end juntos.

A última pasta de exemplo no repositório com a qual trabalharemos é a `projetos/05-testando-aplicacoes-de-ponta-a-ponta` e é nela que desenvolveremos nossos testes de ponta a ponta.

Essa pasta está vazia de código-fonte porque, geralmente, esses testes também são feitos dentro das próprias aplicações. Como deixamos essa categoria como uma parte separada neste livro, faz bastante sentido mantermos essa estrutura no repositório também.

14.1 Como funciona um teste E2E

Um teste end-to-end (ou E2E, ou também funcional) tende a testar um sistema por completo. Isso é feito ao realizar alguns testes diretamente com um navegador, seja ele visível (de forma que você, enquanto desenvolve, consegue ver os testes sendo executados) ou de forma *Headless* (onde o navegador e os testes são executados, mas sem interface gráfica).

Ao executar esse navegador, as ferramentas de teste end-to-end podem navegar pelas páginas de um sistema. Dessa forma, é

possível simular as ações que qualquer pessoa de fato faria ao interagir com o software desenvolvido.

A ferramenta que utilizaremos para esses testes se chama Cypress, que já está instalada no projeto (https://www.cypress.io/). Mas existem algumas outras no mercado que também vou deixar como conteúdo extra ao final do livro.

Para que possamos realizar os testes juntos, apague a pasta `cypress` e o arquivo `cypress.json`, que estão na raiz.

14.2 Conhecendo o Cypress

Para executar o Cypress pela primeira vez é só rodar o comando `npm run test:e2e:open` (que executará `cypress open`). Isso exibirá algumas mensagens no terminal e, logo após, uma janela abrirá com uma interface mais ou menos assim:

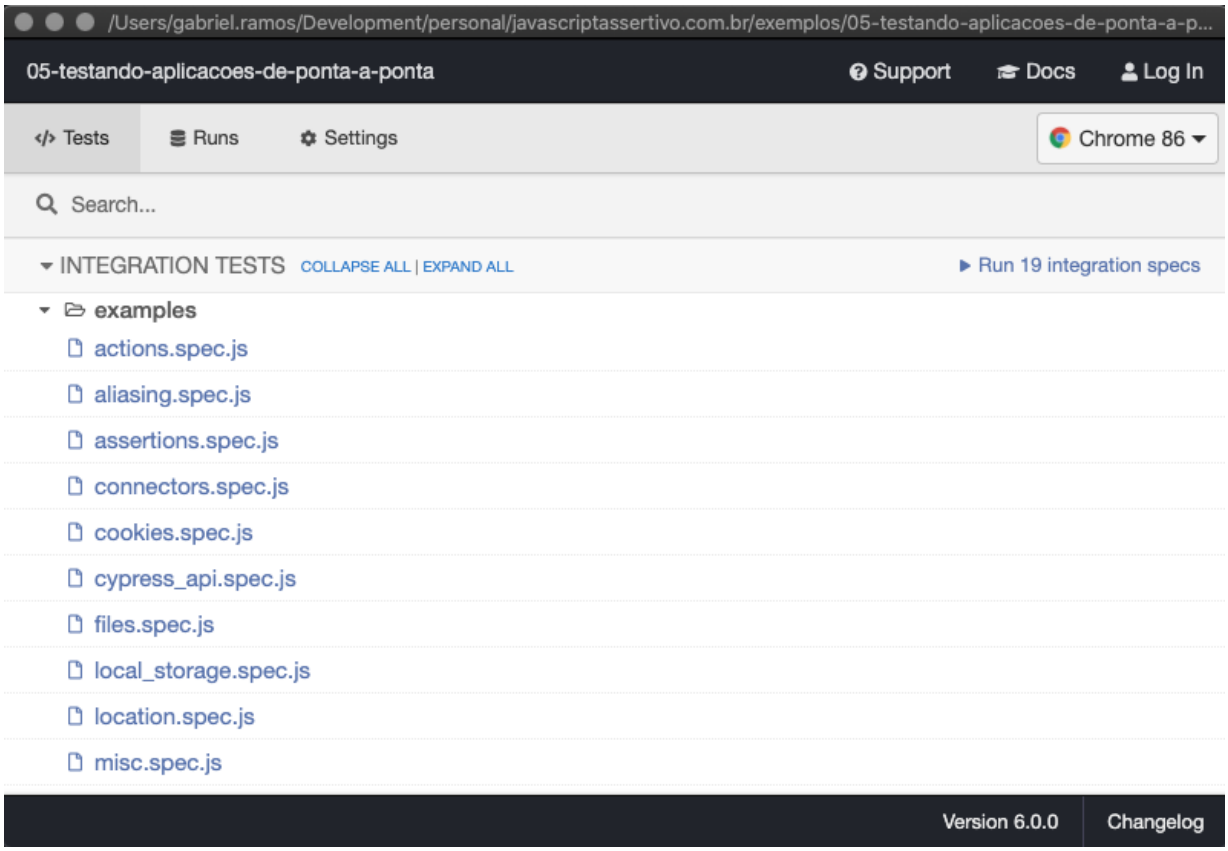


Figura 14.1: Tela inicial do Cypress.

Ao executar esse comando (e, consequentemente, nesse diretório), será criada uma pasta `cypress` (exatamente a que foi removida) com alguns arquivos para que qualquer pessoa, como nova usuária de Cypress, possa entender como ele funciona.

Dentro dessa pasta, algumas outras foram criadas com alguns testes, alguns arquivos de `fixtures` (dados que são usados ao trafegar as requisições) e outras coisas.

Dentro dessa janela, existirão alguns testes de exemplo. Você consegue ver cada um deles, (dentro da pasta `examples`), executá-los (ao clicar no botão `Run 19 integration specs`) e é possível até mesmo simular diferentes navegadores (por padrão, o que aparece selecionado para mim é o Chrome 86). É possível também verificar algumas configurações extras sobre o ambiente de testes.

Pegue algum tempo para rodar esse teste e familiarizar-se um pouco com Cypress. Você verá que, ao clicar na opção de executar os testes (`Run 19 integration specs`), um navegador similar ao Chrome será aberto e os testes começarão a rodar em sequência. Você pode, inclusive, interagir com esse navegador e até mesmo utilizar seu console!

Após isso, apague o conteúdo das pastas `fixtures` , `integration` e `plugins` , removendo qualquer teste existente para que possamos fazer tudo do zero. Antes de iniciar nosso código, vamos realizar uma pequena configuração para auxiliar nossos testes.

14.3 Criando um teste inicial

Vamos criar um arquivo `teste.js` na pasta `integration` . Faremos um teste bem simples onde vamos apenas acessar a página `javascriptassertivo.com.br` .

Assim como as estruturas que já conhecemos no Jest, é possível criar blocos com `describe` / `it` e até realizar asserções com Cypress! Vamos lá:

```
// arquivo cypress/integration/teste.js
describe('Acessa uma página', () => {
  it('Site do livro', () => {});
});
```

Para que possamos navegar para essa página, dentro do nosso bloco `it` , podemos executar a função `cy.visit('')` fornecendo como argumento a url a ser visitada:

```
describe('Acessa uma página', () => {
  it('Site do livro', () => {
    cy.visit('javascriptassertivo.com.br');
```

```
  });
});
```

Agora, rode o comando `npm run test:e2e:open` caso não tenha rodado ainda e, na tela do Cypress, abra esse arquivo de teste que acabamos de criar. Você verá que, ao fazer isso, um navegador igual ao Chrome será aberto e nosso teste funcionará exatamente como queríamos: conseguimos acessar uma página utilizando Cypress!

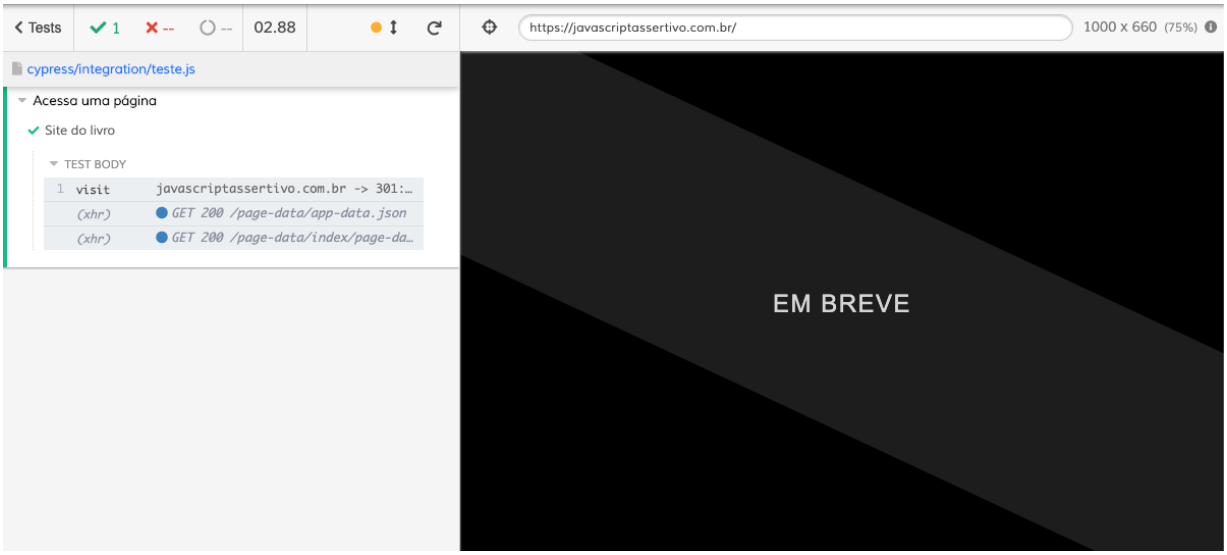


Figura 14.2: Primeiro teste.

Nessa interface, do lado esquerdo, é possível acompanhar os testes em uma espécie de linha do tempo. Conforme você passa o mouse pelos blocos em `test body` é possível verificar a aplicação naquele momento exato em que o teste rodou. Já no lado direito, é possível ver ao vivo o resultado que o navegador está executando.

Esse primeiro teste é importante para entendermos como a estrutura do Cypress vai funcionar. Agora, vamos criar alguns utilitários para facilitar nosso trabalho.

14.4 Criando alguns comandos com a API do Cypress

Precisaremos executar algumas ações repetitivas em nosso teste, como acessar a tela da aplicação e realizar um login.

Para acessar uma página, acabamos de ver que a função `cy.visit` pode nos ajudar, mas podemos facilitar um pouco o trabalho de ficar escrevendo a url da nossa aplicação em todos os testes. Podemos criar nossas próprias abstrações para nos ajudar com algumas tarefas repetitivas.

Vamos criar duas funções, uma como `cy.goToApp` para navegarmos até a aplicação e outra como `cy.login` para realizarmos o login como um usuário.

A própria API do Cypress permite que criemos esses utilitários. Dentro da pasta `cypress/support`, no arquivo `commands.js`, vamos criar um comando chamado `goToApp`. Para fazer isso, é só executar `Cypress.Commands.add('goToApp')`:

```
// arquivo cypress/support/commands.js
Cypress.Commands.add('goToApp');
```

Agora, como segundo argumento dessa função `add`, vamos passar uma função de callback que será executada quando chamarmos `cy.goToApp` em nosso teste. Essa função executará exatamente nosso `cy.visit` na url da aplicação:

```
// inserimos a função de callback
Cypress.Commands.add('goToApp', () => {
  // inserimos o cy.visit para acessar a aplicação
  cy.visit('http://localhost:8081');
});
```

Vamos voltar ao nosso teste de exemplo e vamos executar essa função. Lembre-se de que, para rodarmos os testes com sucesso, as aplicações (tanto front-end quanto back-end) devem estar

rodando corretamente. Existe um outro comando `npm run start:all` nesse projeto do Cypress que executa as duas aplicações para você. Deixe-as executando em um terminal separado.

No nosso teste, vamos executar essa função:

```
describe('Acessa uma página', () => {
  it('Site do livro', () => {
    // alteramos para goToApp()
    cy.goToApp();
  });
});
```

Ao salvar, você verá que o teste já será atualizado e agora a tela exibida é, de fato, a tela da aplicação:

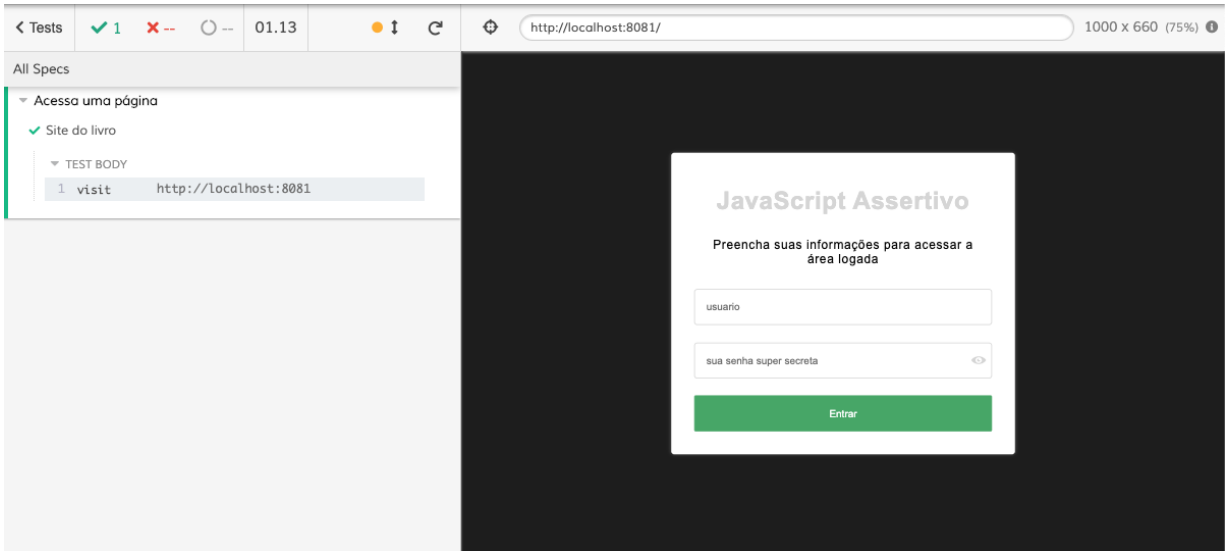


Figura 14.3: Acessando o app com goToApp.

Vamos criar uma função que vai realizar o login para nós, nesse mesmo arquivo de comandos. Vamos chamá-la de `login` e, na função de callback, vamos receber um parâmetro `role` para que possamos realizar o login como usuário comum ou admin:

```
Cypress.Commands.add('login', (role) => {
});
```

Agora, vamos utilizar um usuário/senha real da nossa base de dados. Poderíamos simular as respostas da API utilizando as `fixtures` (https://docs.cypress.io/api/commands/fixture.html), mas, se tratando de um teste que visa testar nossa aplicação de ponta a ponta, podemos deixá-lo acessando nossa aplicação. Em cenários ideais, teríamos um ambiente e uma base de testes especificamente para isso.

Podemos usar o usuário padrão admin (usuário `admin` e senha `admin`) para acessar os dados com uma role de administrador e utilizar qualquer outro usuário com role `USER` para acessar como usuário comum. Colocarei a seguir os dados de algum usuário qualquer. Vamos criar uma variável para armazenar o dado de nome de usuário e outra para a senha, dependendo da role do usuário:

```
Cypress.Commands.add('login', (role) => {
  // variável admin apenas para ser utilizada nos ternários abaixo
  const admin = role.toUpperCase() === 'ADMIN';
  // variável user e password
  const user = admin ? 'admin' : 'Leola_Kautzer74';
  const password = admin ? 'admin' : '5mYFQeV4KQLx9m4';
});
```

Com isso, podemos utilizar esses valores de `user` e `password` para preencher os campos em tela. Podemos utilizar a função `cy.get` para selecionar um elemento em tela. Vamos utilizá-la para selecionar os inputs com seus placeholders:

```
Cypress.Commands.add('login', (role = 'USER') => {
  const admin = role.toUpperCase() === 'ADMIN';
  const user = admin ? 'admin' : 'Leola_Kautzer74';
  const password = admin ? 'admin' : '5mYFQeV4KQLx9m4';
  // adicionamos cy.get para selecionar os dois elementos
  cy.get('input[placeholder="usuario"]');
  cy.get('input[placeholder="sua senha super secreta"]');
});
```

Agora podemos encadear as ações de escrita usando o retorno das funções `cy.get` , através da função `type` , que recebe o valor a ser

digitado. Executaremos a função `type` com a variável `user` no primeiro campo e, no segundo, com a variável `password`:

```
Cypress.Commands.add('login', (role = 'USER') => {
  const admin = role.toUpperCase() === 'ADMIN';
  const user = admin ? 'admin' : 'Leola_Kautzer74';
  const password = admin ? 'admin' : '5mYFQeV4KQLx9m4';
  // adicionamos .type(user);
  cy.get('input[placeholder="usuario"]').type(user);
  // adicionamos .type(password);
  cy.get('input[placeholder="sua senha super secreta"]').type(password);
});
```

Tudo o que precisamos fazer é clicar no botão de entrar. Para selecionar um elemento por seu texto, podemos utilizar a função `cy.contains`:

```
Cypress.Commands.add('login', (role = 'USER') => {
  const admin = role.toUpperCase() === 'ADMIN';
  const user = admin ? 'admin' : 'Leola_Kautzer74';
  const password = admin ? 'admin' : '5mYFQeV4KQLx9m4';

  cy.get('input[placeholder="usuario"]').type(user);
  cy.get('input[placeholder="sua senha super secreta"]').type(password);
  // inserimos cy.contains
  cy.contains('Entrar');
});
```

E agora, encadear uma função `.click` para clicar no elemento:

```
Cypress.Commands.add('login', (role = 'USER') => {
  const admin = role.toUpperCase() === 'ADMIN';
  const user = admin ? 'admin' : 'Leola_Kautzer74';
  const password = admin ? 'admin' : '5mYFQeV4KQLx9m4';

  cy.get('input[placeholder="usuario"]').type(user);
  cy.get('input[placeholder="sua senha super secreta"]').type(password);
  // inserimos click no elemento
  cy.contains('Entrar').click();
});
```

Vamos voltar ao nosso teste e, logo após a função `goToApp`, executar a função `login`:

```
// arquivo cypress/integration.teste.js
describe('Acessa uma página', () => {
  it('Site do livro', () => {
    cy.goToApp();
    // inserimos login
    cy.login();
  });
});
```

Ao inserir essa linha, você verá que o teste será atualizado e um login como usuário comum será realizado. Caso você passe `admin` como parâmetro para a função `login`:

```
describe('Acessa uma página', () => {
  it('Site do livro', () => {
    cy.goToApp();
    // fornecendo 'admin'
    cy.login('admin');
  });
});
```

Você verá que o teste simulará um admin autenticado.

Para finalizar, vamos criar mais uma função chamada `logout`, que clicará no elemento com seletor `[aria-label="sair"]`:

```
// criamos função de logout
Cypress.Commands.add('logout', () => {
  // clicamos no elemento
  cy.get('[aria-label="sair"]').click();
});
```

14.5 Testando o cenário de login

Já criamos dois utilitários que vão facilitar bastante nossos testes. Vamos alterar o nome do arquivo `teste.js` para `login.js` e validar alguns cenários de login.

Faremos algumas adaptações no nosso `describe` / `it` deste arquivo também:

```
// ajustamos describe
describe('Realiza o login', () => {
  // ajustamos it
  it('Como admin', () => {
    cy.goToApp();
    cy.login('admin');
  });
});
```

Primeiro, vamos validar que é possível realizar o login como admin. Após a função `cy.login` vamos executar a função `cy.url()` :

```
describe('Realiza o login', () => {
  it('Como admin', () => {
    cy.goToApp();
    cy.login('admin');

    cy.url();
  });
});
```

Agora vamos validar se a url possui a rota de `/dashboard` . Existem algumas formas de realizar uma asserção no Cypress. Para esse cenário, podemos encadear a função `should` passando os argumentos `include` e `/dashboard` , para validar que a url inclui o caminho da tela de dashboard após o login:

```
describe('Realiza o login', () => {
  it('Como admin', () => {
    cy.goToApp();
    cy.login('admin');

    cy.url().should('include', '/dashboard');
```

```
  });
});
```

E podemos realizar o `logout` também chamando a função `cy.logout` :

```
describe('Realiza o login', () => {
  it('Como admin', () => {
    cy.goToApp();
    cy.login('admin');

    cy.url().should('include', '/dashboard');
    cy.logout();
  });
});
```

Dessa forma, validamos que a url inclui o caminho correto. Vamos duplicar esse teste para o cenário de usuário e remover somente o argumento `admin` da função login:

```
describe('Realiza o login', () => {
  it('Como admin', () => {
    cy.goToApp();
    cy.login('admin');

    cy.url().should('include', '/dashboard');
    cy.logout();
  });

  // duplicamos o teste
  it('Como usuário', () => {
    cy.goToApp();
    cy.login();

    cy.url().should('include', '/dashboard');
    cy.logout();
  });
});
```

Assim validamos que tanto o cenário de usuário padrão quanto o de admin estão conseguindo realizar o login e o logout.

Vale lembrar que, caso algo não ocorra como o esperado no teste, por exemplo, não seja possível navegar até a página da aplicação, isso já faria nossos testes falharem. Não precisamos necessariamente de uma asserção. Realizar os fluxos de uma aplicação já serve para que o teste E2E entregue uma certa segurança.

14.6 Testando as funcionalidades de dashboard

Nesta seção, vamos testar as funcionalidades do painel de perfis. Vamos começar criando o arquivo `dashboard.js` dentro da pasta em que já estávamos criando o arquivo de login.

É possível utilizar os mesmos hooks que já vimos no Jest. Vamos colocar um hook `beforeEach`, que fará com que, em nossos testes, a navegação até a aplicação e o login como admin sejam feitos:

```
beforeEach(() => {
  cy.goToApp();
  cy.login('admin');
});
```

Como vamos cadastrar um usuário, você pode utilizar a função `createUser`, exportada pela CLI, para ajudar nessa tarefa.

É possível acessá-la pelo módulo `'@jsassertivo/cli/commands/user'`:

```
// importamos a função para criar usuário
import { createUser } from '@jsassertivo/cli/commands/user';

beforeEach(() => {
  cy.goToApp();
  cy.login('admin');
});
```

Agora vamos, de fato, cadastrar esse perfil! Vamos criar nosso `it` e, logo de início, vamos clicar no botão que possui o texto

"cadastrar" em seu `aria-label`:

```
import { createUser } from '@jsassertivo/cli/commands/user';

beforeEach(() => {
  cy.goToApp();
  cy.login('admin');
});

// criamos nosso it
it('Cadastra um usuário', () => {
  // clicamos no botão com aria-label correto
  cy.get('[aria-label="cadastrar"]').click();
});
```

Após isso, vamos criar um novo perfil executando a função `createUser`:

```
it('Cadastra um usuário', () => {
  cy.get('[aria-label="cadastrar"]').click();
  // criamos a variável profile com o retorno de createUser
  const profile = createUser();
});
```

Agora, podemos usar os dados de perfil para preencher essa modal. Vamos selecionar os elementos por seus placeholders:

```
it('Cadastra um usuário', () => {
  cy.get('[aria-label="cadastrar"]').click();

  const profile = createUser();

  // selecionamos o email e preenchemos o email do perfil
  cy.get('[placeholder="email"]').type(profile.email);
  // selecionamos o usuario e preenchemos o usuario do perfil
  cy.get('[placeholder="usuario"]').type(profile.userName);
  // selecionamos a senha e preenchemos a senha do perfil
  cy.get('[placeholder="sua senha super
secreta"]').type(profile.password);
  // selecionamos o nome e preenchemos o nome do perfil
  cy.get('[placeholder="nome"]').type(profile.name);
```

```
  // selecionamos o sobrenome e preenchemos o sobrenome do perfil
  cy.get('[placeholder="sobrenome"]').type(profile.lastName);
  // selecionamos a role e preenchemos a role do perfil
  cy.get('[placeholder="role"]').select(profile.role)
});
```

Por fim, podemos apenas clicar no botão de confirmar, exatamente como fizemos no login:

```
it('Cadastra um usuário', () => {
  cy.get('[aria-label="cadastrar"]').click();

  cy.get('[placeholder="email"]').type(profile.email);
  cy.get('[placeholder="usuario"]').type(profile.userName);
  cy.get('[placeholder="sua senha super
secreta"]').type(profile.password);
  cy.get('[placeholder="nome"]').type(profile.name);
  cy.get('[placeholder="sobrenome"]').type(profile.lastName);
  cy.get('[placeholder="role"]').select(profile.role)
  // clicamos no botão
  cy.contains('Confirmar').click();
});
```

Com isso, o cadastro será realizado.

Vamos agora testar a remoção de um usuário. Vamos criar um outro `it`:

```
// novo it
it('Deleta um usuário', () => {
});
```

Como já estamos realizando o login no nosso `beforeEach`, tudo o que precisamos fazer é consultar o botão de deletar no último item do nosso carrossel.

Podemos selecionar o elemento pelo seu `aria-label` como já fizemos anteriormente. Vamos armazená-lo em uma variável:

```
it('Deleta um usuário', () => {
  // selecionamos o botão
```

```
  const button = cy.get('[aria-label="deletar"]');
});
```

Para que possamos pegar somente o último elemento, podemos encadear nossa função `get` com a função `last`:

```
it('Deleta um usuário', () => {
  // utilizamos .last para pegar somente o último
  const button = cy.get('[aria-label="deletar"]').last();
});
```

Agora precisamos clicar nesse elemento. Podemos tentar clicar nele com `button.click`:

```
it('Deleta um usuário', () => {
  const button = cy.get('[aria-label="deletar"]').last();
  // resultará em erro
  button.click();
});
```

Porém, com isso, teremos o seguinte erro:

```
Timed out retrying: cy.click() failed because this element is not visible:
```

Esse erro indica que não foi possível clicar no elemento pelo fato de ele não estar visível. Como esse perfil criado fica no carrossel, que possui uma rolagem horizontal, ele não é visível quando a tela é carregada. Mais um ponto para o Cypress! Com isso, podemos ter certeza que existe uma validação que verifica se quem acessa nossa página pode interagir com nossos elementos.

Para resolver esse problema, temos duas soluções:

- Passar um objeto de configuração como `{ force: true }` ao executar o click, que forçaria a execução da ação mesmo o elemento não sendo visível;
- Realizar o scroll até o elemento com a função `scrollIntoView`, aguardar sua finalização e depois clicar.

Acredito que a segunda solução é mais próxima à forma como uma pessoa interagiria com nossa página, então vamos optar por essa.

Para realizar o scroll até o elemento, basta executarmos um `button.scrollIntoView()`:

```
it('Deleta um usuário', () => {
  const button = cy.get('[aria-label="deletar"]').last();
  button.scrollIntoView();
});
```

Agora, podemos aguardar e inclusive realizar uma asserção verificando que o botão será visível:

```
it('Deleta um usuário', () => {
  const button = cy.get('[aria-label="deletar"]').last();
  button.scrollIntoView();
  // adicionamos asserção que verifica a visibilidade
  button.should('be.visible');
});
```

Depois podemos realizar o click:

```
it('Deleta um usuário', () => {
  const button = cy.get('[aria-label="deletar"]').last();
  button.scrollIntoView();
  button.should('be.visible');
  // clicamos
  button.click();
});
```

E pronto, já temos nosso teste de cadastrar/remover um usuário funcionando perfeitamente! Deixo como um último desafio para você realizar o teste de edição de cadastro.

Executando o Cypress de maneira headless

É possível executar os testes sem necessariamente abrir o navegador. Você pode fazer isso com o comando `npm run test:e2e` (que executará `cypress run`). Isso realizará os testes da mesma forma que ao abrir e simular um navegador, mas sem toda a parte visual do processo.

Com isso, será exibido um relatório do teste com os cenários que passaram ou que falharam, bem parecido com os outros tipos de testes que vimos até o momento.

Vale comentar que, ao final desse processo, um script será executado para "limpar" a base de dados também (é basicamente um `git checkout` no arquivo). Caso tenha curiosidade de ver, ele está configurado no `package.json` e é o arquivo `commands/clear-database.sh` .

Outras configurações e possibilidades do Cypress

Se você notou, alguns arquivos de vídeo foram gerados ao executar os testes end-to-end. É possível desabilitar essa configuração e, inclusive, tirar screenshots das telas da aplicação enquanto os testes estão ocorrendo.

A documentação do Cypress (https://docs.cypress.io/) é um prato cheio e muito bem detalhada. Boa parte das necessidades podem ser configuradas diretamente no arquivo `cypress.json` na raiz da aplicação.

Caso tenha interesse, existe um plugin da própria Testing Library (https://testing-library.com/docs/cypress-testing-library/intro) para utilização com Cypress, facilitando a consulta aos elementos. Para utilizá-la, basta instalá-la no projeto e adicionar seu import no arquivo de comandos em que mexemos anteriormente.

O que testar, quando pensamos em E2E?

Geralmente, essa é uma pergunta que surge após estudar tantas formas de testar e, como muitas coisas dentro do mundo da tecnologia, a resposta é: **depende**.

Como vimos, executar um teste E2E demanda uma certa configuração, já que precisamos simular um navegador e nossa aplicação real deve estar pronta para ser executada. Isso faz com

que um ambiente (em CI/CD) seja necessário, o que também impacta em alguns custos de infraestrutura (e tempo) adicionais.

Com testes unitários e integrados coerentes, a camada de teste de ponta a ponta é boa para testar fluxos "rompendo barreiras", que testes de integração e unitários não rompem, e simular um cenário mais real de utilização da aplicação.

Vai do seu projeto e do seu time decidir quais áreas valem a pena receberem essa cobertura de testes. Geralmente os fluxos mais críticos acabam sendo os mais válidos.

14.7 Terminamos nossa jornada, mas o assunto não para por aqui

Terminamos o conteúdo técnico do livro. No entanto, qualidade de software e garantir confiança nas suas entregas não é algo que para por aqui.

Para finalizar com chave de ouro, vamos comentar sobre tópicos que podem guiar seus estudos daqui para a frente e citar algumas outras ferramentas que podem ajudar seus testes em diversas vertentes.

Parte 6: Extras e conteúdos relevantes após a nossa jornada

Nossa jornada foi longa, mas não acaba por aqui!

Podemos comentar sobre alguns tópicos e conceitos extras que podem guiar seus estudos com testes após a finalização deste livro.

Capítulo 15
Próximos passos nessa jornada

Vale a pena comentar sobre alguns assuntos, ferramentas, tópicos e boas práticas que podem ser aplicadas nos testes (e nas aplicações) após tudo o que vimos até agora.

Testes são um assunto muito extenso por si só e embora nossa jornada tenha sido recheada de conceitos, tenho certeza de que os nossos estudos não param por aqui.

15.1 Ferramentas adicionais e referências

Algumas tecnologias específicas foram utilizadas ao longo do livro, no entanto não são as únicas ferramentas que existem para cada um dos problemas que encontramos.

A seguir, deixo uma lista de todas as ferramentas que utilizamos, junto com várias outras que podem ser úteis para seus testes em diversos casos.

Fora isso, comentamos também alguns materiais desenvolvidos pelo Kent C. Dodds, que virou referência sobre testes na

comunidade. Ele possui um curso muito bom (em inglês) disponível no site https://testingjavascript.com/.

Testes unitários em geral

- Jest (https://jestjs.io/)
- Mocha (https://mochajs.org/)
- Chai (https://www.chaijs.com/)
- Jasmine (https://jasmine.github.io/)
- Karma (https://karma-runner.github.io/latest/index.html)
- Sinon (https://sinonjs.org/)

Renderização de componentes

- Testing Library (https://testing-library.com/)
- Enzyme (https://enzymejs.github.io/enzyme/)

Mocks e configurações para HTTP

- Nock (https://github.com/nock/nock)
- MSW (https://github.com/mswjs/msw)
- SuperTest (https://github.com/visionmedia/supertest)
- PollyJS (https://netflix.github.io/pollyjs/#/README)
- Mockoon (https://mockoon.com/)
- JSON-server (https://github.com/typicode/json-server)

Testes de carga

- Artillery (https://artillery.io/)
- k6 (https://k6.io/)
- AutoCannon (https://github.com/mcollina/autocannon)

Regressão visual

- Loki (https://loki.js.org/)
- Percy (https://percy.io/)
- Backstop (https://github.com/garris/BackstopJS)

- Chromatic (https://www.chromatic.com/)
- Screener (https://screener.io/)

Testes E2E

- Selenium (https://www.selenium.dev/documentation/en/webdriver/)
- Puppeteer (https://pptr.dev/)

15.2 Você já ouviu falar em TDD, BDD e DDD?

São algumas práticas, estruturas e comportamentos (se é que podemos chamar assim) que têm como objetivo guiar uma certa conduta de desenvolvimento de software. Vamos dar uma breve olhada no que cada uma delas significa. Podem ser tópicos interessantes para seus estudos futuros.

O que é TDD

Significa *Test-Driven Development*, ou desenvolvimento orientado a testes. É uma cultura de desenvolvimento que visa escrever software guiado pelos testes.

Nessa cultura, o processo de desenvolvimento é comumente conhecido como *Red, Green, Refactor*, que indicam as três etapas para um determinado desenvolvimento, onde:

- Red: indica o cenário inicial, onde você escreve um teste antes de escrever uma linha de código sequer, o que claramente indicará que seu teste vai falhar;
- Green: você escreve a implementação necessária para que seu teste passe;
- Refactor: você refatora a sua implementação aplicando boas práticas necessárias e deixando seu código mais coerente com as necessidades da aplicação.

Práticas de TDD são encorajadas pelo fato de gerarem um código coeso. Geralmente, começamos o desenvolvimento pensando diretamente em nossa implementação, mas mudar esse pensamento e deixar os testes guiarem suas implementações é algo bem diferente e que leva seu código a implementações, muitas vezes, mais simples.

Eu particularmente assumo que tenho bastante dificuldade em exercer TDD e ainda tenho um caminho longo de exercícios pela frente. Também acho que existem cenários onde praticar TDD se torna muito difícil, principalmente quando você acaba de entrar em um projeto e não possui domínio nenhum sobre o código e o contexto necessário.

Em outros cenários, como no início de um projeto do zero ou até mesmo quando você encontra algum bug (e, consequentemente, deve escrever um teste garantindo que o bug não vai se repetir), já é mais simples de se aplicar TDD.

O que é BDD

Significa *Behavior-Driven Development*, ou desenvolvimento orientado a comportamentos. Como o nome sugere, técnicas de BDD visam escrever estruturas relacionadas aos comportamentos (e requisitos) que uma aplicação deve possuir.

Por exemplo, se fôssemos aplicar uma estrutura de BDD no nosso teste de cadastro de usuário, poderíamos ter algo como:

```
Funcionalidade: Cadastrar usuário
    Para quando novo usuário existir
    Eu, como administrador
    Desejo cadastrar um novo perfil

Cenário: Cadastra um novo perfil
    Dado que estou logado como ADMIN
    E possuo os dados de usuário
    Quando eu clicar em cadastrar
```

```
    E preencher os dados
    Então o perfil deve ser criado
```

Isso é uma estrutura de escrita em BDD (em português). Na minha opinião, o grande ganho desse tipo de estrutura é a clareza no que deve ocorrer em determinado sistema. Claro que nossas estruturas `describe/it` já deixam os cenários de teste bem claros, mas, ainda assim, a estrutura que acabamos de ver deixa tudo muito mais claro.

Essa estrutura, em específico, assemelha-se muito à estrutura da ferramenta chamada Gherkin (https://cucumber.io/docs/gherkin/reference/), fornecida pela Cucumber (https://cucumber.io/), uma ferramenta de BDD bem famosa.

É possível integrá-la a algumas outras de teste, como o Cypress (https://www.npmjs.com/package/cypress-cucumber-preprocessor), permitindo que você escreva testes em uma estrutura BDD fugindo totalmente do padrão `describe` / `it` , que aprendemos.

15.3 Comentários sobre decisões e tópicos que não foram o nosso foco

Durante todo o livro, nosso foco foi direcionado exclusivamente para os testes e para entender como aplicá-los de forma mais eficiente em nosso software.

Por isso, algumas decisões e abordagens que os projetos utilizam estão longe de serem as melhores quando pensamos em uma aplicação real que iria para um ambiente de produção e teria contato com clientes.

De decisões estruturais a alguns fluxos, acho que vale a pena comentar alguns deslizes propositais que foram cometidos para que

focássemos nos testes.

Estrutura de pastas, nomenclaturas e organização de arquivos de teste

Ao longo de todos os projetos, os testes foram centralizados em uma pasta específica. Nem sempre isso será o padrão para seus projetos. Se você notou, ao criar uma pasta `__tests__` , tivemos que, a todo instante, ficar replicando a estrutura de diretórios e arquivos do nosso código-fonte para que essas estruturas ficassem semelhantes.

É comum em muitos projetos os arquivos de testes ficarem juntos aos códigos que estão sendo testados, agilizando, em muitos casos, o desenvolvimento e manutenção dos testes.

Vamos lembrar o cenário dos componentes da aplicação de front-end. Para cada componente, tivemos que criar um novo arquivo com o padrão `[componente].unit.js` dentro `__tests__/components` . Se esses testes ficassem dentro dos diretórios dos próprios componentes (junto aos arquivos de definição, estilos e stories), o desenvolvimento dessa aplicação no dia a dia seria um pouco mais prático, já que cada pasta de cada componente conteria todos os seus arquivos relacionados. Mas, para quem acompanha o material do livro é muito mais fácil, ao clonar o projeto, apagar somente uma pasta do que apagar diversos arquivos em pastas separadas.

Um outro detalhe é sobre as nomenclaturas `.unit` e `.integration` . Foi interessante utilizá-las ao longo do livro para que as diferenças entre testes de unidade e de integração ficassem bem claras. Entretanto, nomenclaturas como `arquivo.test.js` ou `arquivo.spec.js` são também muito utilizadas.

Essas decisões foram tomadas para facilitar nossa jornada, mas cada projeto e cada necessidade pode adotar uma estrutura diferente.

Disponibilidade e compartilhamento de código entre pacotes

Como vimos em alguns capítulos anteriores, as aplicações lidam diretamente umas com as outras. Isso é feito de algumas maneiras:

- Ao instalá-las como um pacote através do seu caminho (por isso o `file://` no arquivo `package.json`);
- Ao executar o comando `node` diretamente em outro diretório.

Essas abordagens não são as melhores para aplicações e compartilhamento de módulos em um ambiente real. É comum realizar a publicação de pacotes compartilhados no próprio registro do NPM (https://www.npmjs.com/).

Embora seja possível instalar e executar as aplicações dessa forma (e existam algumas outras https://gabrieluizramos.com.br/maneiras-incomuns-de-instalar-um-pacote-npm), optar por seguir com essa estrutura nos poupou algum tempo de configuração no ambiente e alguns esforços de publicação e versionamento dos projetos envolvidos.

Autenticação, segurança e informações sensíveis

Você provavelmente notou que toda a parte de autenticação da aplicação não é das melhores e das mais seguras. Não existe um gerenciamento muito coerente de login e, inclusive, as informações sensíveis, como senha e `uid`, que são usadas para autenticação, são trafegadas ao longo de todo o sistema. Isso foi uma escolha voluntária enquanto os projetos foram desenvolvidos.

Como nosso foco é aprender a testar aplicações, deixamos de lado quesitos de segurança para que pudéssemos focar no que importa para esse momento. Afinal, as aplicações desenvolvidas aqui não possuem qualquer intuito de irem para o ar e serem utilizadas de verdade, são apenas exemplos didáticos para aplicarmos testes da melhor forma possível.

Claramente essa não é a melhor forma de se fazer um sistema de autenticação de forma segura. Todo o software desenvolvido como objeto de estudo neste livro teve um foco bem simples: aplicar conceitos fundamentais de tecnologia para que o máximo de cenários pudessem ser cobertos com os mais variados tipos de testes.

Mocks e bases de teste

Em muitos testes (exceto os E2E) optamos por sempre realizar o mock de ao menos a última camada da nossa aplicação, que iria interagir com a "base de dados" e salvar os perfis no arquivo `database.json`.

Isso foi uma escolha completamente viável e que faz muito sentido para grande parte das aplicações. Entretanto, em alguns projetos pode ser que você possua algumas bases em um ambiente de teste de forma que alguma camada acaba, de fato, lendo ou escrevendo algo em algum banco.

Embora isso possa acontecer, vai depender bastante do ambiente em que você está trabalhando.

Automação e fluxo de CI/CD

Cada projeto, cada empresa e cada estrutura vai possuir uma particularidade quando o assunto é CI/CD e, como isso é algo bem particular, ao longo de nossa jornada aprendemos apenas a realizar os testes e os conceitos que eles envolvem.

Com certeza, o ideal não é manter o fluxo de testes manual, de forma que você precise executar os comandos de teste toda vez que fizer alguma modificação em uma base de código. No entanto, a tarefa de automatizar esses procedimentos em sua infraestrutura não foge muito de tudo o que aprendemos aqui.

15.4 Obrigado por tudo e até breve

Espero que ao longo de todo esse trajeto você tenha aprendido e se divertido com tudo o que testamos.

Independente da camada, da aplicação, da estrutura e do tipo de teste que você escolher e puder colocar em prática, desejo que você consiga aplicar com sucesso os fundamentos que vimos aqui e com tudo isso gerar confiança na aplicação que você escreve, entregando sempre software com a melhor qualidade possível.

Lembre-se de que, caso precise, estou em (praticamente) todas as redes sociais através do perfil *@gabrieluizramos*. Você pode encontrar essas redes sociais (e também meus posts) no meu site https://gabrieluizramos.com.br/ e também todo o conteúdo relacionado ao livro no site https://javascriptassertivo.com.br/ sempre que precisar.

Será uma honra bater um papo sobre testes com você ou tirar qualquer dúvida sobre algo que possa ter ficado em aberto por aqui. Qualquer coisa é só chamar!

Que seus testes entreguem confiança e qualidade no código que você escreve. Até mais!

Capítulo 16
Glossário

Esqueceu o que algum termo significa no contexto de testes? Quer revisar algo? Esta lista pode lhe servir de auxílio:

- CLI: sigla para *command-line interface*, programas que são executados através de uma interface via linha de comando (terminal).
- Cobertura (coverage): percentual (%) de um código que está ou não validado (e, portanto, coberto) por testes.
- Declarações (statements): verificam se todas as declarações de código (criações de variáveis, chamadas de funções, blocos de código em geral) foram executadas ao longo dos testes.
- Fakers: trecho de dado responsável por simular algum conteúdo, como uma estrutura de dados utilizada dentro de alguma função testada.
- Funções (functions): verificam se as funções presentes em um código foram ou não executadas no decorrer dos testes.
- Git: ferramenta de versionamento de código.
- GitHub: site/nuvem (ou hub) onde repositórios de versionamento de código são compartilhados, por isso o nome Git + Hub.
- Limite (threshold): limite válido de aceitação de cobertura de um teste, por exemplo, um código novo deve ter uma cobertura de, no mínimo, 80% (de linhas, declarações, branches e funções).
- Linhas (lines): linhas de código testadas.
- Mocks: ferramenta responsável por simular um comportamento esperado (seja de uma função ou de algum dado);
- NodeJS: plataforma utilizada para execução de códigos JavaScript localmente em uma máquina (conhecido também como "no servidor").
- NPM: sigla para Node Package Manager, ou Gerenciador de Pacotes do Node, é a ferramenta responsável por instalar e

lidar com os pacotes reutilizáveis, nos ambientes em que NodeJS é utilizado.

- NPX: sigla para Node Package Execute, ou Executor de Pacotes do Node, bem similar ao NPM, é uma ferramenta responsável por apenas executar alguns pacotes em projetos ou globalmente;
- Ramificações (branches): variações de fluxo dentro de um teste como, por exemplo, if/else;
- Relatório (report): informações e gráficos contendo o estado do código escrito com a cobertura (e, consequentemente as ramificações, linhas e declarações) definida de testes.
- Spies: ferramenta responsável por observar o comportamento de alguma função (espionar).
- Stubs: ferramenta responsável por substituir um comportamento de alguma função ou trecho de código.